REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

(2)

AD-A226 897

REPORT DATE	3. REPORT TYPE AND DATES COVERED
	Final 11 June 1990 to 11 June 1991

4. TITLE AND SUBTITLE Ada Compiler Validation Summary Report: Telesoft TeleGen2 Sun-3 Ada Development System, Sun-3 (Host & Target), 900525I1.11012

5. FUNDING NUMBERS

6. AUTHOR(S)
IABG-AVF
Ottobrunn, FEDERAL REPUBLIC OF GERMANY

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
IABG-AVF, Industrieanlagen-Betriebsgeselschaft
Dept. SZT
Einsteinstrasse 20
D-8012 Ottobrunn
FEDERAL REPUBLIC OF GERMANY

8. PERFORMING ORGANIZATION REPORT NUMBER
AVF-IABG-070

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Joint Program Office
United States Department of Defense
Washington, D.C. 20301-3081

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

TeleSoft, TeleGen2 Sun-3 Ada Development System, Ottobrunn, West Germany, Sun Microsystems Sun-3 Workstation (68020 based Sun-3/280) under Sun UNIX 4.2, Release 4.0.3, ACVC 1.11.

14. SUBJECT TERMS Ada programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, Validation Testing, Ada Validation Office, Ada Validation Facility, ANSI/MIL-STD-1815A, Ada Joint Program Office

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

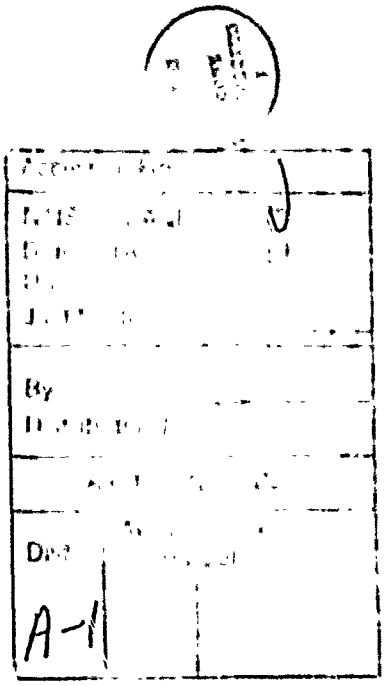
Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation-dependent but is permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

AVF Control Number: AVF-IABG-070
11 June 1990

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: #900525I1.11012
Telesoft
TeleGen2 Sun-3 Ada Development System
Sun-3 Host and Target

Prepared By:
IABG mbH, Abt. YTE
Einsteinstrasse 20
D-8012 Ottobrunn
West Germany



Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 25 May 1990.

Compiler Name and Version: TeleGen2 Sun-3 Ada Development System, Version 4.01

Host Computer System: Sun Microsystems Sun-3 Workstation (68020 based Sun-3/280) under Sun UNIX 4.2, Release 4.0.3

Target Computer System: Same as host

A more detailed description of this Ada implementation is found in section 3.1 of this report.

As a result of this validation effort, Validation Certificate [redacted] is awarded to Telesoft. This certificate expires on 01 June 1992.

This report has been reviewed and is approved.

Michael Tonndorf

IABG mbH, Abt. ITE
Michael Tonndorf
Einsteinstrasse 20
D-8012 Ottobrunn
West Germany

Ada Validation Organization
Director, Computer & Software
Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

Jon Edward Kelleher II

Ada Joint Program Office
Dr. John Solomond
Director
Department of Defense
Washington DC 20301

DECLARATION OF CONFORMANCE

Customer: TeleSoft

Ada Validation Facility: IABG, Dept. SZT, D-8012 Ottobrunn

ACVC Version: 1.11

Ada Implementation:

Compiler Name and Version: TeleGen2™ Sun-3 Ada Development System, Version 4.01

Host Computer System: Sun Microsystems® Sun-3 Workstation
(68020 based Sun-3/280)
Sun™ UNIX™ 4.2, Release 4.0.3 Operating System

Target Computer System: same as Host Computer System

Customer's Declaration

I, the undersigned, representing TeleSoft, declare that TeleSoft has no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A in the implementation listed in this declaration.

Date: May 22, 1990

TeleSoft
Raymond A. Parra, Vice President and General Counsel

Sun Microsystems is a registered trademark of Sun Microsystems, Inc. Sun is a trademark of Sun Microsystems, Inc. UNIX is a trademark of AT&T/Bell Laboratories.

CONTENTS

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro89] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro89]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which preformed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.2 REFERENCES

[Ada83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

[Pro89] Ada Compiler Validation Procedures, Version 2.0, Ada Joint Program Office, May 1989.

[UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values -- for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1) and, possibly some inapplicable tests (see Section 3.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.
Conformity	Fulfillment by a product, process or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer	A computer system where Ada source programs are transformed

System	into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro89].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 65 tests had been withdrawn by the Ada Validation Organization (AVO) when preparing for validation testing. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 90-03-23.

E28005C	C34006D	B41308B	C45114A	C45612B	C45651A
C46022A	B49008A	A74006A	B83022B	B83022H	B83025B
B83025D	B83026B	C83026A	C83041A	C97116A	C98003B
BA2011A	CB7001A	CB7001B	CB7004A	CC1223A	BC1226A
CC1226B	BC3009B	CD2A21E	CD2A23E	CD2A32A	CD2A41A
CD2A41E	CD2A87A	CD2B15C	BD3006A	CD4022A	CD4022D
CD4024B	CD4024C	CD4024D	CD4031A	CD4051D	CD5111A
CD7004C	ED7005D	CD7005E	AD1B08A	AD7006A	CD7006E
AD7201A	AD7201E	CD7204B	BD8002A	BD8004C	CD9005A
CD9005B	CDA201E	CE2107I	CE2119B	CE3111C	CE3116A
CE3411B	CE3412B	CE3812A	CE3814A	CE3902B	

Tests B28006C and C43004A were withdrawn before the start of validation. The tests were processed and the results were ignored.

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. The inapplicability criteria for some tests are explained in documents issued by ISO and the AJPO known as Ada Issues and commonly referenced in the format AI-dddd. For this implementation, the following tests were inapplicable for the reasons indicated; references to Ada Issues are included as appropriate.

The following 201 tests have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)

C35708L..Y (14 tests) C35802L..Z (15 tests)
C45241L..Y (14 tests) C45321L..Y (14 tests)
C45421L..Y (14 tests) C45521L..Z (15 tests)
C45524L..Z (15 tests) C45621L..Z (15 tests)
C45641L..Y (14 tests) C46012L..Z (15 tests)

C35404D, C45231D, B86001X, C86006E, and CD7101G check for a predefined integer type with a name other than INTEGER, LONG_INTEGER, or SHORT_INTEGER.

C35508I, C35508J, C35508M, and C35508N include enumeration representation clauses for BOOLEAN types in which the representation values are other than (FALSE => 0, TRUE => 1). Under the terms of AI-00325, this implementation is not required to support such representation clauses.

C35702A, C35713B, C45423B, B86001T, and C86006H check for the predefined type SHORT_FLOAT.

C35713D and B86001Z check for a predefined floating-point type with a name other than FLOAT or LONG_FLOAT.

C45531M..P (4 tests) and C45532M..P (4 tests) are not applicable because they require a value of SYSTEM.MAX_MANTISSA greater than 32.

C45624A checks that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types with digits 5. For this implementation, MACHINE_OVERFLOWS is TRUE.

C45624B checks that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types with digits 6. For this implementation, MACHINE_OVERFLOWS is TRUE.

C86001F recompiles package SYSTEM, making package TEXT_IO, and hence package REPORT, obsolete. For this implementation, the package TEXT_IO is dependent upon package SYSTEM.

B86001Y checks for a predefined fixed-point type other than DURATION.

CA2009C, CA2009F, BC3204C and BC3205D make use of certain separate compilation features for generic units which are not supported by this implementation. This implementation creates a dependence between a generic body and those units which instantiate it. As allowed by AI-408/11, if the body is compiled after a unit that instantiates it, then that unit becomes obsolete.

CD1009C uses a representation clause specifying a non-default size for a floating-point type.

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use representation clauses specifying non-default sizes for access types.

The tests listed in the following table are not applicable because the given file operations are supported for the given combination of mode and file access method.

Test	File Operation	Mode	File Access Method
CE2102D	CREATE	IN_FILE	SEQUENTIAL_IO
CE2102E	CREATE	OUT_FILE	SEQUENTIAL_IO
CE2102F	CREATE	INOUT_FILE	DIRECT_IO
CE2102I	CREATE	IN_FILE	DIRECT_IO
CE2102J	CREATE	OUT_FILE	DIRECT_IO
CE2102N	OPEN	IN_FILE	SEQUENTIAL_IO
CE2102O	RESET	IN_FILE	SEQUENTIAL_IO
CE2102P	OPEN	OUT_FILE	SEQUENTIAL_IO
CE2102Q	RESET	OUT_FILE	SEQUENTIAL_IO
CE2102R	OPEN	INOUT_FILE	DIRECT_IO
CE2102S	RESET	INOUT_FILE	DIRECT_IO
CE2102T	OPEN	IN_FILE	DIRECT_IO
CE2102U	RESET	IN_FILE	DIRECT_IO
CE2102V	OPEN	OUT_FILE	DIRECT_IO
CE2102W	RESET	OUT_FILE	DIRECT_IO
CE3102E	CREATE	IN_FILE	TEXT_IO
CE3102F	RESET	Any Mode	TEXT_IO
CE3102G	DELETE	--------	TEXT_IO
CE3102I	CREATE	OUT_FILE	TEXT_IO
CE3102J	OPEN	IN_FILE	TEXT_IO
CE3102K	OPEN	OUT_FILE	TEXT_IO

CE2107B..E (4 tests), CE2107L, and CE2110B attempt to associate multiple internal files with the same external file when one or more files is writing for sequential files. The proper exception is raised when multiple access is attempted.

CE2107G..H (2 tests), CE2110D, and CE2111H attempt to associate multiple internal files with the same external file when one or more files is writing for direct files. The proper exception is raised when multiple access is attempted.

CE2111D checks the resetting of an external file from IN_FILE to OUT_FILE.

CE2203A checks that WRITE raises USE_ERROR if the capacity of the external file is exceeded for SEQUENTIAL_IO. This implementation does not restrict file capacity.

CE2403A checks that WRITE raises USE_ERROR if the capacity of the external file is exceeded for DIRECT_IO. This implementation does not restrict file capacity.

CE3111B, CE3111D..E (2 tests), CE3114B, and CE3115A attempt to associate multiple internal files with the same external file when one or more files is writing for text files. The proper exception is raised when multiple access is attempted.

CE3304A checks that USE_ERROR is raised if a call to SET_LINE_LENGTH or SET_PAGE_LENGTH specifies a value that is inappropriate for the external file. This implementation does not have inappropriate values for either line length or page length.

CE3413B checks that PAGE raises LAYOUT_ERROR when the value of the page

number exceeds COUNT'LAST. For this implementation, the value of COUNT'LAST is greater than 150000 making the checking of this objective impractical.

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 21 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

BA1001A	BA2001C	BA2001E	BA3006A	BA3006B
BA3007B	BA3008A	BA3008B	BA3013A	

The following 4 tests cause errors at bind time instead of at compile time.

BC3204C	BC3205D	CA2009C	CA2009F

CD1009A, CD1009I, CD1C03A, CD2A21C, CD2A24A, CD2A31A..C (3 tests), use instantiations of the support procedure Length_Check, which uses Unchecked_Conversion according to the interpretation given in AI-00590. The AVO ruled that this interpretation is not binding under ACVC 1.11; the tests are ruled to be passed if they produce Failed messages only from the instantiations of Length_Check -- ie, the allowed Report.Failed messages have the general form:

" * CHECK ON REPRESENTATION FOR <TYPE_ID> FAILED."

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For a point of contact for both technical and sales information about this Ada implementation system, see:

Telesoft
5959 Cornerstone Court West
San Diego, CA 92121-9819
(619) 457-2700

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

3.2 TEST EXECUTION

Version 1.11 of the ACVC comprises 4606 tests. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors. The AVF determined that 280 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation. In addition, the modified tests mentioned in section 2.3 were also processed.

A tape cartridge containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the tape cartridge were loaded onto an auxiliary computer and then transferred via Ethernet to the host computer.

After the test files were loaded onto the host computer, the full set of tests was processed by the Ada implementation.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options.

Tests were compiled using the command

```
ada  -O  2  <file name>
```

Tests were linked using the command

```
ald  <main unit>
```

The -L qualifier was added to the compiler call for class B, expanded and modified tests. The -m <main unit> qualifier was added to the compiler call for single non class B tests.

Test output, compiler and linker listings, and job logs were captured on tape cartridge and archived at the AVF. The listings examined on-site by the validation team were also archived.

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The following macro parameters are defined in terms of the value V of $MAX_IN_LEN which is the maximum input line length permitted for the tested implementation. For these parameters, Ada string expressions are given rather than the macro values themselves.

Macro Parameter	Macro Value
$BIG_ID1	(1..V-1 => 'A', V => '1')
$BIG_ID2	(1..V-1 => 'A', V => '2')
$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
$BIG_INT_LIT	(1..V-3 => '0') & "298"
$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
$BIG_STRING1	'"' & (1..V/2 => 'A') & '"'
$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & '"'
$BLANKS	(1..V-20 => ' ')
$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
$MAX_STRING_LITERAL	'"' & (1..V-2 => 'A') & '"'

The following table contains the values for the remaining macro parameters.

Macro Parameter	Macro Value
$MAX_IN_LEN	200
$ACC_SIZE	32
$ALIGNMENT	4
$COUNT_LAST	2_147_483_646
$DEFAULT_MEM_SIZE	2147483647
$DEFAULT_STOR_UNIT	8
$DEFAULT_SYS_NAME	TELEGEN2
$DELTA_DOC	2#1.0#E-31
$ENTRY_ADDRESS	ENT_ADDRESS
$ENTRY_ADDRESS1	ENT_ADDRESS1
$ENTRY_ADDRESS2	ENT_ADDRESS2
$FIELD_LAST	1000
$FILE_TERMINATOR	' '
$FIXED_NAME	NO_SUCH_TYPE
$FLOAT_NAME	NO_SUCH_TYPE
$FORM_STRING	""
$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
$GREATER_THAN_DURATION	100_000.0
$GREATER_THAN_DURATION_BASE_LAST	131_073.0
$GREATER_THAN_FLOAT_BASE_LAST	1.80141E+38
$GREATER_THAN_FLOAT_SAFE_LARGE	1.80141E+38
$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	0.0

```
$HIGH_PRAGMA1           93

$ILLEGAL_EXTERNAL_FILE_NAME1
                        BADCHAR*^/%

$ILLEGAL_EXTERNAL_FILE_NAME2
                        NONAME/DIRECTORY

$INAPPROPRIATE_LINE_LENGTH
                        -1

$INAPPROPRIATE_PAGE_LENGTH
                        -1

$INCLUDE_PRAGMA1        PRAGMA INCLUDE ("A28006D1.TST")

$INCLUDE_PRAGMA2        PRAGMA INCLUDE ("B28006F1.TST")

$INTEGER_FIRST          -32768

$INTEGER_LAST           32767

$INTEGER_LAST_PLUS_1    32768

$INTERFACE_LANGUAGE     C

$LESS_THAN_DURATION     -100_000.0

$LESS_THAN_DURATION_BASE_FIRST
                        -131_073.0

$LINE_TERMINATOR        ASCII.LF

$LOW_PRIORITY           0

$MACHINE_CODE_STATEMENT
                        MCI'(OP=>NOP);

$MACHINE_CODE_TYPE      Opcodes

$MANTISSA_DOC           31

$MAX_DIGITS             15

$MAX_INT                2147483647

$MAX_INT_PLUS_1         2147483648

$MIN_INT                -2147483648

$NAME                   NO_SUCH_TYPE_AVAILABLE

$NAME_LIST              TELEGEN2
```

$NAME_SPECIFICATION1	/usr6wayward/gint/rh4.01_111_12_val/ceb/X2120A
$NAME_SPECIFICATION2	/usr6wayward/gint/rh4.01_111_12_val/ceb/X2120B
$NAME_SPECIFICATION3	/usr6wayward/gint/rh4.01_111_12_val/cec/X3119A
$NEG_BASED_INT	16#FFFFFFFE#
$NEW_MEM_SIZE	2147483647
$NEW_STOR_UNIT	8
$NEW_SYS_NAME	TELEGEN2
$PAGE_TERMINATOR	ASCII.FF
$RECORD_DEFINITION	RECORD NULL; END RECORD;
$RECORD_NAME	NO_SUCH_MACHINE_CODE_TYPE
$TASK_SIZE	32
$TASK_STORAGE_SIZE	2048
$TICK	0.02
$VARIABLE_ADDRESS	VAR_ADDRESS
$VARIABLE_ADDRESS1	VAR_ADDRESS1
$VARIABLE_ADDRESS2	VAR_ADDRESS2
$YOUR_PRAGMA	COMMENT EXPORT IMAGES INTERFACE_INFORMATION INTERRUPT LINKNAME NO_SUPPRESS PRESERVE_LAYOUT SUPPRESS_ALL TASK_INFORMATION TASK_TRANSFORMATION

APPENDIX B

COMPILATION SYSTEM OPTIONS

The compiler options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

LINKER OPTIONS

The linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to linker documentation and not to this report.

2
Compilation Tools

This chapter discusses the commands to invoke the TeleGen2 components that are associated with the process of compilation. The components are the compiler (invoked by the *ada* command; see Section 2.1) and the linker (invoked by the *ald* command; see Section 2.2).

Optimization is part of the compilation process as well. In the TeleGen2 documentation set, however, optimization is discussed separately from compilation. In this volume, the commands associated with optimization (*ada -O*; *aopt*) are discussed in the "Other Tools" chapter. (One exception is the Option Summary table below, where *aopt* options are included for comparison.)

Table 2-1 summarizes the options that are used by the compilation tools. Note that several options are common to the commands shown.

Table 2-1. Compilation Tools Option Summary

Option	Command		
	ada	*ald*	*aopt*
-l(ibfile	x	x	x
-t(emplib	x	x	x
-V(space_size	x	x	x
-v(erbose	x	x	x
-b(ind_only		x	
-C(ontext	x		
-d(ebug	x		
-E(rror_abort	x		
-e(rrors_only	x		
-F(ile_only_errs	x		
-G(raph	x		x
-I(nline	x		x
-i(nhibit	x		
-j(oin	x		
-k(eep	x		x
-K(eep	x		
-L(ist	x		
-m(ain	x		
-n(on_ada_bind		x	
-N(ame			x
-O(ptimize	x		x
-o(utput		x	
-R(ecomp_minimization	x		
-S[a]	x	x	x
-u(pdate_lib	x		
-w("timeslice"		x	
-X(ception_show		x	
-x(ecution_profile	x	x	x
-Y		x	

Note

a: The functionality of the -S option of *ada* and the -S option of *ald* is somewhat different. Refer to the text.

2.1. The Ada Compiler ("ada")

The TeleGen2 Ada Compiler is invoked by the *ada* command. Unless you specify otherwise, the front end, middle pass, and code generator are executed each time the compiler is invoked.

Before you can compile, you must (1) make sure you have access to TeleGen2, (2) create a library file, and (3) create a sublibrary. These steps were explained in the Getting Started section of the Overview. We suggest you review that section, and then compile, link, and execute the sample program as indicated before you attempt to compile other programs.

This section focuses specifically on command-level information relating to compilation, that is, on invoking the compiler and using the various options to control the compilation process. Details on the TeleGen2 compilation process and guidelines for using the compiler most effectively are in the Compiler chapter of the *User Guide* volume. (You might want to look at Figure 3-1 in that volume right now, to give you insight into the TeleGen2 compilation process and to see how the options mentioned in this Command Summary volume relate to the actual compilation process.)

The syntax of the command to invoke the Ada compiler is:

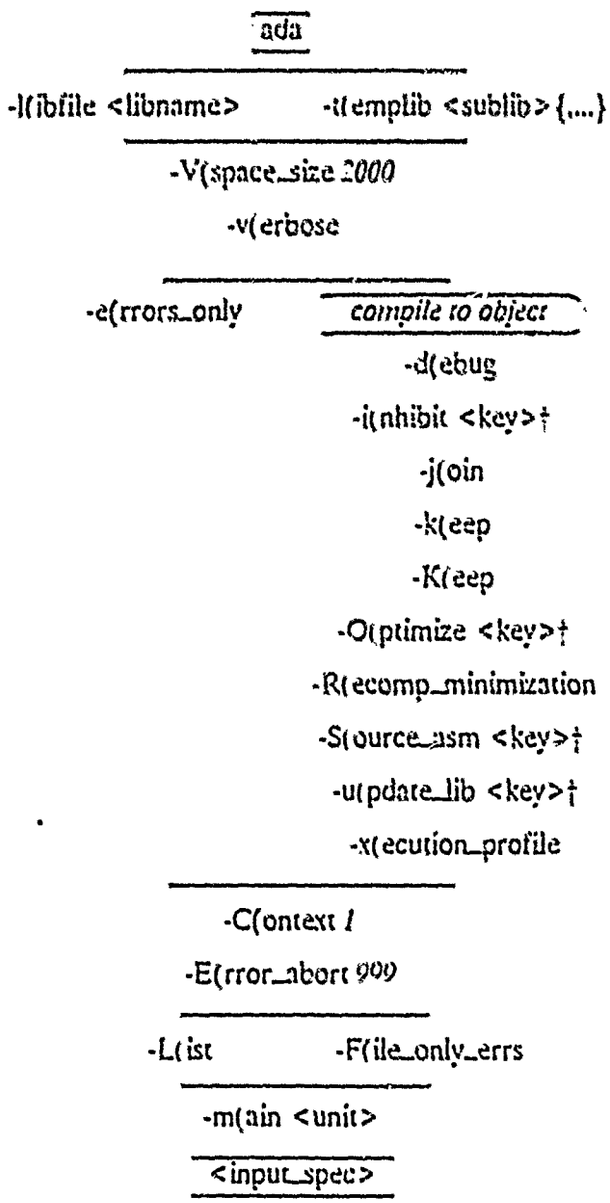
```
ada {<"common_option">} {<option>} <input_spec>
```

where:

<"common_option">	None or more of the following set of options that are common to many TeleGen2 commands: -l(ibfile or -t(emplib -V(space_size -v(erbose These options were discussed in Chapter 1.
<option>	None or more of the compiler-specific options discussed below.
<input_spec>	The Ada source file(s) to be compiled. It may be: • One or more Ada source files, for example: /user/john/example Prog_A.text ciosrc/calc_mem.ada calcio.ada myprog.ada *.ada • A file containing names of files to be compiled. Such a file must have the extension ".ilf". You can find details for using input-list files in the *User Guide* portion of your TeleGen2 documentation set. • A combination of the above.

Please note that the compiler defaults are set for your convenience. In most cases you will not need to use additional options; a simple "ada <input_spec>" is sufficient. However, options are included to provide added flexibility. You can, for example, have the compiler quickly check the source for syntax and semantic errors but not produce object code (-e(rrors_only) or you can compile, bind, and link an main program with a single compiler invocation (-m(ain). Other options are provided for other purposes.

The options available with the *ada* command, and the relationships among them, are illustrated in the following figure. Remember that each of the options listed is identified by a dash followed by a single letter (e.g., "-e") The parenthesis and the characters following the option are for descriptive purposes only; they are not part of the option.



† (1) <key> for -O: refer to *dopt*. (2) <key> for -a: f or s; s is the default. (2) key for -a or certain combinations of inc. (3) <key> for -S: e or n; n is the default.

The options available with the *ada* command are summarized in Table 2-2. The default situation (that is, what happens if the option is not used) is explained in the middle column. Each option is described in the paragraphs that follow the table.

Table 2-2. Summary of Compiler Options

Option	Default	Discussed in Section
Common options:		
-l(ibfile <libname>	Use liblst.alb as the library file.	1.4.1
-t(emplib <sublib...>	None	1.4.1
-V(space_size <value>	Set size to 2000 Kbytes.	1.4.2
-v(erbose	Do not output progress messages.	1.4.3
-d(ebug	Do not include debug information in object code. (*-d* sets -k(eep.)	2.1.1
-E(rror_abort <value>	Abort compilation after 999 errors.	2.1.2
-e(rrors_only	Run middle pass and code generator, not just front end.	2.1.3
-i(nhibit <key>†	Do not suppress run-time checks, source line references, or subprogram name information in object.	2.1.4
-j(oin	Do not join the errors with the source file.	2.1.5
-k(eep	Discard intermediate representations of secondary units.	2.1.6
-K(eep	Discard the source file and do not store it in the library.	2.1.7
-m(ain <unit>	Do not produce executable code (binder/linker not executed).	2.1.8
-O(ptimize <key>†	Do not optimize code.	2.1.9
-R(ecomp_minimization	Recompile completely; dependent units may become obsolete.	2.1.10
-u(pdate_lib <key>†	Do not update library when errors are found (multi-unit compilations).	2.1.11
-x(ecution_profile	Do not generate execution-profile code.	2.1.12

† (1) <key> for -O: refer to *aopt*. (2) <key> for -r: 1 or s, s is the default. (3) <key> for i: a or certain combinations of inc.

Table 2-2. Summary of Compiler Options (Continued)

Option	Default	Discussed In Section
Listing options:		
-C(ontext <value>	Include 1 line of context with error message.	2.1.13.1
-L(ist	Do not generate a source-error listing.	2.1.13.2
-F(ile_only_errs	Do not generate an errors-only listing. only.	2.1.13.3
-S(ource_asm <key>†	Do not generate assembly listing.	2.1.13.4

2.1.1. -d(ebug - Generate Debugger Information

The code generator must generate special information for any unit that is to be used with the TeleGen2 symbolic debugger. The generation of this information is enabled by use of the *-d* option. The use of *-d* automatically sets the -k(eep option. This to make sure that the High Form, Low Form, and debugger information for secondary units are not deleted.

To see if a unit has been compiled with the -d(ebug option, use the *als* command with the -X(tended option. Debugger information exists for the unit if the "dbg_info" attribute appears in the listing for that unit. The default situation is that no debugger information is produced.

Performance note. While the compilation time overhead generated by the use of -d(ebug is minimal, retaining this optional information in the Ada library increases the space overhead.

2.1.2. -E(rror_abort - Set an Error Count for Aborting Compilation

The compiler maintains separate counts of all syntactic errors, semantic errors, and warning messages detected by the front end during a compilation.

A large number of errors generally indicates that errors in statements appearing earlier in the unit have "cascaded" through the rest of the code. A classic example is an error occurring in a statement that declares a type. This causes subsequent declarations that use the type to be in error, which further causes all statements using the declared objects to be in error. In such a situation, only the first error message is useful. Aborting the compilation at an early stage is therefore often to your advantage; the *-E* option allows you to do it.

† (1) <key> for S: e or n; n is the default.

The format of the option is:

-E <value>

where <value> is the number of errors or warnings allowed. The default value is 999. The minimum value is 1. Caution: If you do not use the *-E* option, it is possible to have 999 warning messages *plus* 999 syntax errors *plus* 999 semantic errors without aborting compilation, since each type of error is counted separately.

2.1.3. -e(rrors_only - Check Source But Don't Generate Code

This option instructs the compiler to perform syntactic and semantic analysis of the source program without generating Low Form and object code. That is, it calls the front end only, not the middle pass and code generator. (This means, of course, that only front end errors are detected and that only the High Form intermediates are generated.) This option is typically used during early code development where execution is not required and speed of compilation is important.

Note: Although High Form intermediates are generated with the *-e* option, these intermediates are deleted at the end of compilation. This means that the library is not updated.

The *-e* option cannot be used with -S(ource_asm, since the latter requires the generation of object code. If *-e* is not used (the default situation), the source is compiled to object code (if no errors are found). The *-e* option is also incompatible with -k(eep, -d(ebug, -O(ptimize, and other options that require processing beyond the front end phase of compilation.

2.1.4. -i(nhibit - Suppress Checks and Source Information

The -i(nhibit option allows you to suppress, within the generated object code, certain run-time checks, source line references, and subprogram name information.

The Ada language requires a wide variety of *run-time checks* to ensure the validity of operations. For example, arithmetic overflow checks are required on all numeric operations, and range checks are required on all assignment statements that could result in an illegal value being assigned to a variable. While these checks are vital during development and are an important asset of the language, they introduce a substantial overhead. This overhead may be prohibitive in time-critical applications.

Although the Ada language provides pragma Suppress to selectively suppress classes of checks, using the pragma requires you to modify the Ada source. The -i(nhibit option provides an alternative mechanism.

The compiler by default stores *source line and subprogram name information* in the object code. This information is used to display a source level traceback when an unhandled exception propagates to the outer level of a program; it is particularly valuable during development, since it provides a direct indication of the source line at which the exception occurs and the subprogram calling chain that led to the line generating the exception.

The inclusion of source line information in the object code, however, introduces an overhead of 6 bytes for each line of source that causes code to be generated. Thus, a 1000-line package may have up to 6000 bytes of source line information. For one compilation unit, the extra overhead (in bytes) for subprogram name information is the total length of all subprogram names in the unit (including middle pass-generated subprograms), plus the length of the compilation unit name. For space-critical applications, this extra space may be unacceptable; but it can be suppressed with the -i(nhibit option. When source line information is suppressed, the traceback indicates the offset of the object code at which the exception occurs instead of the source line number. When subprogram name information is suppressed, the traceback indicates the offsets of the subprogram calls in the calling chain instead of the subprogram names. (For more information on the traceback function, refer to the Programming Guide chapter in your *Reference Information* volume.)

The format of the -i(nhibit option is:

-i <suboption>{<suboption>}

where <suboption> is one or more of the single-letter suboptions listed below. Combinations of suboptions are possible. When more than one suboption is used, the suboptions appear together with no separators. For example, "-i lnc".

l[line_info]	Suppress source line information in object code.
n[ame_info]	Suppress subprogram name information in object code.
c[hecks]	Suppress run-time checks -- elaboration, overflow, storage access, discriminant, division, index, length, and range checks.
a[ll]	Suppress source line information, subprogram name information, and run-time checks. In other words, a (=inhibit all) is equivalent to lnc.

As an example of use, the command...

```
ada -v -i lc my_file.ada
```

...inhibits the generation of source line information and run-time checks in the object code of the units *my_file.ada*.

2.1.5. -j(oin - Join Errors with Source Code

This option joins the any errors that are generated with the source file. The resulting file is written back to the source file. The errors appear in the file as Ada comments. This option allows you to comment the source file with the errors that are generated at compile time. These comments can help facilitate debugging and commenting your code.

Unlike the -L and -F options, the -j option does not produce a listing.

2.1.6. -k(eep - Retain Intermediate Forms

As a default, the compiler deletes the High Form and Low Form intermediate representations of all compiled secondary units from the working sublibrary. Deletion of these intermediate forms can significantly decrease the size of sublibraries – typically 50% to 80% for multi-unit programs. On the other hand, some of the information within the intermediate forms may be required later. For example, High Form is required if the unit is to be referenced by the Ada Cross-Referencer (*axr*). In addition, information required by the debugger and the Global Optimizer must be saved if these utilities are used. For these reasons, the *-k* option is provided with the *ada* command. The *-k* option:

- Must be used if the compiled unit is to be optimized later with *aopt*; otherwise, *aopt* issues an error message and the optimizer aborts.
- Should be used if the unit is to be cross-referenced later; otherwise, an error message is issued when the Ada Cross-Referencer attempts to cross-reference that unit.
- Need not be used with -d(ebug, since *-k* is set automatically whenever *-d* is used.

To verify that a unit has been compiled with the -k(eep option (has not been "squeezed"), use the *als* command with the -X(tended option. A listing will be generated that shows whether the intermediate forms for the unit exist. A unit has been compiled with -k(eep if the attributes high_form and low_form appear in the listing for that unit.

2.1.7. -K(eep - Keep Source File in Library

This option tells the compiler to take the source file and store it in the Ada library. When you later need to retrieve your source file, you may use the *aco*

command.

2.1.8. -m(ain - Compile a Main Program

This option tells the compiler that the unit specified with the option is to be used as a main program. After all files named in the input specification have been compiled, the compiler invokes the prelinker (binder) and the native linker by calling *ald* to bind and link the program with its extended family. An executable file named <unit> is left in the current directory. The linker may also be invoked directly by the user with the *ald* command.

The format of the option is:

-m <unit>

where <unit> is the name of the main unit for the program. If the main unit has already been compiled, it does not have to be in the input file. However, the body of the main unit, if previously compiled, must be present in the current working sublibrary.

Note: Options specific to the linker (invoked via *ald*) may be specified on the *ada* command line when the *-m* option is used. With *-m*, the compiler will call *ald* when compilation is complete, passing to it *ald*-specific options specified with the *ada* command. For example...

ada -m welcome -o new sample.ada

...instructs the compiler to compile the Ada source file, *sample.ada*, which contains the main program unit Welcome. After the file has been compiled, the compiler calls the linker, passing to it the *-o* option with its respective arguments. The linker produces an executable version of the unit, placing it in file *new* as requested by the *-o* option.

2.1.9. -O(ptimize - Optimize Object Code

This option causes the compiler to invoke the global optimizer to optimize the Low Form generated by the middle pass for the unit being compiled. The code generator takes the optimized Low Form as input and produces more efficient object code. The format of this option is:

-O <key>

where <key> is at least one of the optimizer suboption keys discussed in the Global Optimizer chapter. Please refer to that chapter for all information regarding the use of the optimizer. The chapter discusses using the optimizer as a standalone tool for collections of compiled but unoptimized units and using the -O(ptimize option with the *ada* command. The latter topic includes a definition of the -O(ptimize suboption key values plus a presentation of two other *ada* options (-G(raph and -I(nline_list, not shown on the *ada* chart) that may be used in conjunction with the -O(ptimize option. *Note*: We strongly recommend that you do not attempt to use the optimizer until the code being compiled has been fully

debugged and tested.

2.1.10. -R(ecomp_minimization - Minimizing Recompilation

This option has the following function eliminates the need for recompilation of dependent units when an "insignificant" change is made to another unit. An "insignificant" change has been defined as one of the following:

1. Adding/Deleting/Modifying a comment.

2. Changing case of identifiers, including reserved words. NOTE: Changing the case of string or character literals including operators is NOT allowed.

3. Changing case of letters within numeric literals, equivalent numeric literals.

4. Reformatting changes (white space).

A restriction of the current implementation is that units dependent upon a unit with an insignificant change that contains inline or instantiation dependencies must always be recompiled.

A dependency of the current implementation is that the source of the compiled unit be stored in the library. This is necessary in order to determine the level of change to the unit being recompiled.

2.1.11. -u(pdate_lib - Update the Working Sublibrary

The -u(pdate_lib option tells the compiler when to update the library. It is most useful for compiling multiple source files. The format of the option is:

-u <key>

where <key> is either "s" (source) or "i" (invocation).

i "i" tells the compiler to update the working sublibrary after all files submitted in that invocation of *ada* have compiled successfully. If an error is encountered, the library is not updated, even for source files that compile successfully. In addition, all remaining source files will be compiled for syntactic and semantic errors only. *Implications:* (1) If an error exists in any source file you submit, the library will not be updated, even if all other files are error free. (2) Compilation is faster, since the library is updated only once, at the end of compilation.

s (This is the default; it is equivalent to not using the -u(pdate_lib option at all.) "s" tells the compiler to update the library after all units within a single source file compile successfully. If the compiler encounters an error in any unit within a source file, all changes to the working sublibrary for the erroneous unit and for all other units in the file are discarded. However, library updates for units in previous or remaining source files are unaffected. *Implications:* (1) You can submit files containing possible errors and still have units in other files compile successfully into the library. (2) Compilation is slightly slower, since the library is updated once for each file.

Therefore:

Use "u s" (or no -u(pdate option) when:

You're not sure all units will compile successfully.
Compilation speed is not especially important.

Use "u i" when:

You are reasonably certain your files will compile successfully.
Fast compilation is important.

2.1.12. -x(ecution_profile - Generate Profile Information

The -x(ecution_profile option uses the code generation phase of compilation to place special information in the generated code that can be used to obtain a "profile" of a program's execution. This information is generated by a facility known as "the profiler." Refer to your *User Guide* volume for information on how to use the profiler to obtain execution timing and subprogram call information for a program.

Important: If any code in a program has been compiled with the -x(ecution_profile option, that option must also be used with *ald* when the program is bound and linked. Otherwise, linking aborts with an error such as "Undefined RECORDSSCURRENT".

2.1.13. Listing Options

The listing options specify the content and format of listings generated by the compiler. Assembly code listings of the generated code can also be generated.

2.1.13.1. -C(ontext - Include Source Lines Around the Error

When an error message is sent to *stderr*, it is helpful to include the lines of the source program that surround the line containing the error. These lines provide a context for the error in the source program and help to clarify the nature of the error. The *-C* option controls the number of source lines that surround the the error.

The format of the option is:

```
-C  <value>
```

where <value> is the number of source context lines output for each error. The default for <value> is 1. This parameter specifies the total number of lines output for each error (including the source line that contains the error). The first context line is the one immediately before the line in error; other context lines are distributed before and after the line in error. Let's say that *trialprog.ada*, which consists of the following text...

```
package P is
  type T1 is range 1..10;
  type T2 is digits 1;
  type Arr is array (1..2) of integer;
  type T3 is new Arr;     -- OK.

    package Inner is
      type In1 is new T1;      -- ERROR: T1 DERIVED.
      type In2 is new T2;      -- ERROR: T2 DERIVED.
      type In3 is new T3;      -- ERROR: T3 DERIVED.
      type Inarr is new Arr;   -- OK.

    end Inner;

end P;
```

...were compiled as follows:

```
ada  -e  -C 2  trialprog.ada
```

(The -e option here is used for error checking and -C(ontext is set to 2 to display two lines of source.) The output produced would look like this:

```
     7:     package Inner is
     8:       type In1 is new T1;      -- ERROR: T1 DERIVED.
                                       ------
>>>    Illegal parent type for derivation <3.4:15,7.4.1:4>


     8:       type In1 is new T1;      -- ERROR: T1 DERIVED.
     9:       type In2 is new T2;      -- ERROR: T2 DERIVED.
                                       ------
>>>    Illegal parent type for derivation <3.4:15,7.4.1:4>


     9:       type In2 is new T2;      -- ERROR: T2 DERIVED.
    10:       type In3 is new T3;      -- ERROR: T3 DERIVED.
                                       ------
>>>    Illegal parent type for derivation <3.4:15,7.4.1:4>
```

2.1.13.2. -L(ist - Generate a Source Listing

This option instructs the compiler to output a listing of the source being compiled, interspersed with error information (if any). The listing is output to <file_spec>.l, where <file_spec> is the name of the source file (minus the extension). If <file_spec>.l already exists, it is overwritten.

If input to the *ada* command is an input-list file (<file_spec>.ilf), a separate listing file is generated for each source file listed in the input file. Each resulting listing file has the same name as the parent file, except that the extension ".l" is appended. Errors are interspersed with the listing. If you do not use -L (the default situation), errors are sent to *stdout* only; no listing is produced. -L is incompatible with -F.

2.1.13.3. -F(ile_only_errs - Put Only Errors in Listing File

This option is used to produce a listing containing only the errors generated during compilation; source is not included. The output is sent to <file_spec>.l. -F is incompatible with -L.

2.1.13.4. -S(ource_asm - Generate a Source/Assembly Listing

This option instructs the compiler to generate an assembly listing and send it to a file named <unit>.s, where <unit> is the name of the unit in the user-supplied source file. (The file extension may be different on your system.) The listing consists of assembly code intermixed with source code as comments. If input to the *ada* command is an input-list file (<file_spec>.ilf), a separate assembly listing file is generated for each unit contained in each source file listed in the input file. If -S is not used (the default situation), an assembly listing is not generated.

The format of this option is:

```
-S <key>
```

where <key> is either "e" (extended) or "n" (normal). The argument to the -S option, <key>, is mandatory.

e "e" tells the compiler to generate a paginated, extended assembly listing which will include code offsets and object code.

n "n" tells the compiler to generate a normal assembly listing. This listing can later be used as input to the Sun assembler.

2.2. The Ada Linker ("ald")

The TeleGen2 Ada Compiler produces object code from Ada source code. The TeleGen2 Ada Linker takes the object (of a main program) that is produced by the compiler and produces a UNIX executable module. The TeleGen2 Ada Linker will be called "the linker" in the remainder of this manual.

To produce executable code, the linker (1) generates elaboration code and a link script (this is called "binding" or "prelinking") then (2) calls the UNIX link editor (*ld*) to complete the linking process.

The linker is invoked with the *ald* command; it can also be invoked with the -m(ain option of the *ada* command. In the latter case the compiler passes appropriate options to the linker, to direct its operation.

In the simplest case, the *ald* command takes one argument - the name of the main unit of the Ada program structure that is to be linked - and produces one output file - the executable file produced by the linking process. The executable file is placed in the directory where *ald* was executed, under the name of the main unit used as the argument to *ald*. (For System V versions of UNIX: if the name is longer than 14 characters, it is truncated.) For example, the command

```
ald  main
```

links the object modules of all the units in the extended family of the unit Main. The name of the resulting executable file will simply be "main". Important: When using the *ald* command, the body of the main unit to be prelinked must be in the working sublibrary.

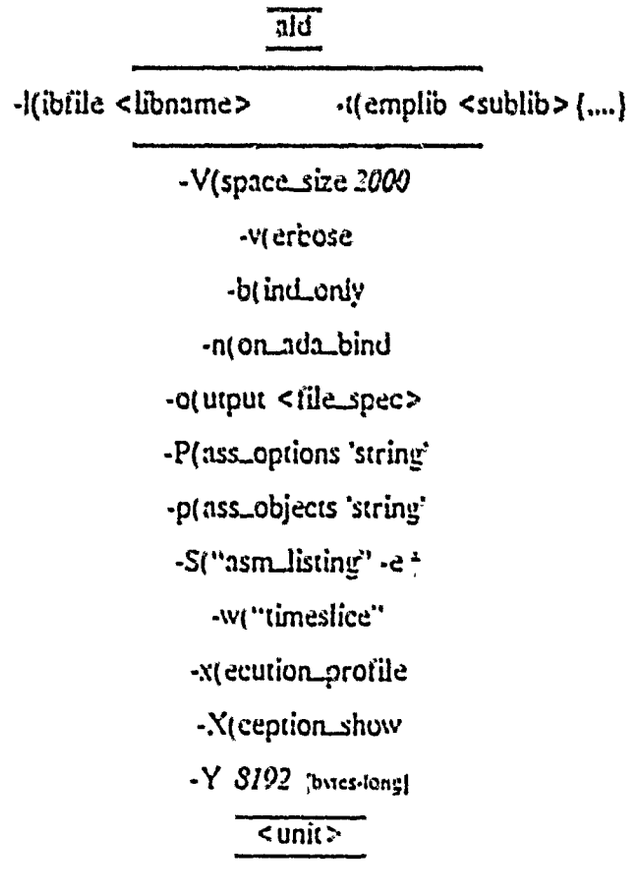
The general syntax of the *ald* command is:

```
ald  {<"common_option">}  {<option>}  <unit>
```

where:

<"common_option">	None or more of the following set of options that are common to many TeleGen2 commands: -l(ibfile or -t(emplib -V(space_size -v(erbose These options were discussed in Chapter 1.
<option>	None or more of the options discussed in the following sections.
<unit>	The name of the main unit of the Ada program to be linked.

The options available with the *ald* command and the relationships among them are illustrated below.



2.2.1. -b(ind_only - Produce Elaboration Code Only

To provide you with more control over the linking process, the *-b* option causes the linker to abort after it has created the elaboration code and the linking order, but before invoking the UNIX link editor. This option allows you to edit the link order for special applications and then invoke the link editor directly. The link order is contained in an executable script that invokes the link editor with the appropriate options and arguments. The name of the script produced is <unit>.lnk, which is placed in your working directory. To complete the link process, enter "<unit>.lnk".

The name of the file containing the elaboration code is <unit>.obm, which is placed in the object directory of the working sublibrary.

For System V versions of UNIX, the file names generated as a result of linking are created by appending the 3-letter extension to the unit name and truncating the result to 14 characters.

2.2.2. -n(on_ada_bind - Non-Ada Main Binding

The -n option causes the binder to produce elaboration code and a link script appropriate for making non-Ada code a main program. Non-Ada object modules can simply be prepended to the Ada object modules and the script can then be executed to produce an image executable under a non-Ada environment.

The format of the option is:

```
-n
```

This is similar to the -b switch which causes the binder to produce elaboration code and a link script appropriate for making Ada code a main program.

2.2.3. -o(utput - Name the Output File

This option allows you to specify the name of the output file produced by the linker. For example, the command...

```
ald  -o yorkshire  main
```

...causes the linker to put the executable module in the file *yorkshire*.

2.2.4. -P(ass_Options - Pass Options to the Linker

This option allows you to pass a string of options directly to the UNIX link editor. For example, the command

```
ald  -P '-t -r'  main
```

adds the string "-t -r" to the options of the link editor when it is invoked. The options must be quoted (double or single quotes).

2.2.5. -p(ass_objects - Pass Arguments to the Linker

This option allows you to pass a string of arguments directly to the UNIX link editor. For example, the command

```
ald -p 'cosine.o /usr/lib/libm.a' main
```

causes the link editor to link the object file *cosine.o* (which it expects to find in the current working directory), and to search the library */usr/lib/libm.a* for unresolved symbol references. (The location of the *libm.a* library may be different on your system.) Remember that the link editor searches a library exactly once at the point it is encountered in the argument list, so references to routines in libraries must occur before the library is searched. That is, files that include references to library routines must appear before the corresponding libraries in the argument list. Objects and archives added with the *-p* option will appear in the linking order after Ada object modules and run-time support libraries, but before the standard C library (/lib/libc.a). This library is always the last element of the linking order.

You can also use the *-p* option to specify the link editor's *-l* option, which causes the link editor to search libraries whose names have the form "/lib/lib*name*.a" or "/usr/lib/lib*name*.a". For example, the command

```
ald -p '-lxyz'
```

causes the link editor to search the directories */lib* and */usr/lib* (in that order) for file *libxyz.a*.

2.2.6. -S("asm_listing" - Produce an Assembly Listing

The *-S* option is used to output an assembly listing from the elaboration process. The output is put in a file, <file>.obm.s, where <file> is the name of the main unit being linked. (The file extension may be different on your system.)

The format of this option is:

```
-S <key>
```

where <key> is either "e" (extended) or "n" (normal). The argument to the -S option, <key>, is mandatory.

- e — "e" tells the compiler to generate a paginated, extended assembly listing which will include code offsets and object code.
- n — "n" tells the compiler to generate a normal assembly listing. This listing can later be used as input to the Sun assembler.

2.2.7. -s(oftware_float - Use Software Floating-Point Support

This option may not be available on your TeleGen2 system. Please consult the Overview portion of this volume to see if it is provided. The Ada linker currently selects hardware floating-point support by default. This default situation is provided for users of systems with an arithmetic coprocessor. If you do not have

hardware floating point support or if you wish to generate code compatible with such machines, use the *-s* option. *In addition*: if you use the *-s* option, the library file you use for compiling and linking must contain the name of the *software* floating point run-time sublibrary, *.sub*. Refer to the Library Manager chapter in your *User Guide* volume for more information on the run-time sublibrary.

2.2.8. -x(ecution_profile - Bind and Link for Profiling

This option is used for units that have been compiled with the *-x* option. Use of *-x* with *ada* causes the code generator to include, in the object, special code that will later be used to provide a profile of the program's execution.

If *-x* is used with *ada*, it must be used with *ald* as well. The *-x* option of *ald* instructs the linker to link in the profiling run-time support routines and generate a subprogram dictionary, *profile.dic*, for the program. The dictionary is a text file containing the names and addresses of all subprograms in the program. The dictionary can be used to produce a listing showing how the program executes.

Refer to the Ada Profiler chapter in your *User Guide* volume for a full discussion of the profiler.

2.2.9. Tasking Options

The following *ald* options are binding options used for task execution. They are therefore useful only for linking programs that contain tasking code.

2.2.9.1. -w("timeslice" - Limit Task Execution Time

The *-w* option allows you to define the maximum time a task may execute before it is rescheduled. The format of the option is:

```
-w <value>
```

where <value> is the maximum time the task is to execute, in milliseconds, before a task switch occurs between it and a task having the same or higher priority. The default value is 0 (no timeslice). If you choose a value greater than 0, it must be at least as great as the clock interval time

Since rescheduling of tasks is incompatible with interrupt-scheduling, *-w* is incompatible with -D(elay_NonPreempt (see above).

2.2.9.2. -X(ception_show - Report Unhandled Exceptions

By default, unhandled exceptions that occur in tasks are not reported; instead, the task terminates silently. The *-X* option allows you to specify that such exceptions are to be reported. The output is similar to that displayed when an unhandled exception occurs in a main program.

2.2.9.3. -Y - Alter Stack Size

In the absence of a representation specification for task storage_size, the run time will allocate $192 bytes of storage for each executing task. You can change the amount of space allocated for tasking by using the -Y option.

-Y specifies the size of the basic task stack. The format of the option is:

-Y <value>

where <value> is the size of the task stack in 32-bit (long_integer) bytes. The default is $192.

A representation specification for task storage size overrides a value supplied with this option.

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD which are not a part of Appendix F, are

```
package STANDARD is

   ...

      type INTEGER is range -32768 .. 32767;
      type SHORT_INTEGER is range -128 .. 127;
      type LONG_INTEGER is range -2147483648 .. 2147483647;

      type FLOAT is digits 6 range -1.70141E+38 .. 1.70141E+38;
      type LONG_FLOAT is digits 15 range -1.79769E+308 .. 1.79769E+308;

      type DURATION is delta 2#1.0#E-14 range -86400.0 .. 86400.0;

   ...

end STANDARD;
```

3
LRM Annotations

TeleGen2 compiles the full ANSI Ada language as defined by the *Reference Manual for the Ada Programming Language* (LRM) (ANSI/MIL-STD-1815A). This chapter describes the portions of the language that are designated by the LRM as implementation dependent for the compiler and run-time environment.

The information is presented in the order in which it appears in the LRM. In general, however, only those language features that are not fully implemented by the current release of TeleGen2 or that require clarification are included. The features that are optional or that are implementation dependent, on the other hand, are described in detail. Particularly relevant are the sections annotating LRM Chapter 13 (Representation Clauses and Implementation-Dependent Features) and Appendix F (Implementation-Dependent Characteristics).

3.1. LRM Chapter 2 - Lexical Elements

[LRM 2.1] Character Set. The host and target character set is the ASCII character set.

[LRM 2.2] Lexical Elements, Separators, and Delimiters. The maximum number of characters on an Ada source line is 200.

[LRM 2.8] Pragmas. TeleGen2 implements all language-defined pragmas *except* pragma Optimize. If pragma Optimize is included in Ada source, the pragma will have no effect. Optimization is implemented by using pragma Inline and the optimizer. Pragma Inline is not supported for library-level subprograms.

Limited support is available for pragmas Memory_Size, Storage_Unit, and System_Name: that is, these pragmas are allowed if the argument is the same as the value specified in the System package.

Pragmas Page and List are supported in the context of source/error listings: refer to the Compiler/Linker chapter of the TeleGen2 *User Guide* for more information.

3.2. LRM Chapter 3 - Declarations and Types

[LRM 3.2.1] Object Declarations.

TeleGen2 does not produce warning messages about the use of uninitialized variables. The compiler will not reject a program merely because it contains such variables.

[LRM 3.5.1] Enumeration Types. The maximum number of elements in an enumeration type is 32767. This maximum can be realized only if generation of the image table for the type has been deferred, and there are no references in the program that would cause the image table to be generated. Deferral of image table generation for an enumeration type, P, is requested by the statement:

```
pragma Images (P, Deferred);
```

Refer to "Implementation-Defined Pragmas." in Section 3.8.1, for more information on pragma Images.

[LRM 3.5.4] Integer Types. There are three predefined integer types: Short_Integer, Integer, and Long_Integer. The attributes of these types are shown in Table 3-1. Note that using explicit integer type definitions instead of predefined integer types should result in more portable code.

Table 3-1. Attributes of Predefined Types Short_Integer, Integer, and Long_Integer

Attribute	Type		
	Short_Integer	Integer	Long_Integer
'First	-128	-32768	-2147483648
'Last	127	32767	2147483647
'Size	8	16	32
'Width	4	6	11

[LRM 3.5.8] Operations of Floating Point Types. There are two predefined floating point types: Float and Long_Float. The attributes of types Float and Long_Float are shown in Table 3-2. This floating point facility is based on the IEEE standard for 32-bit and 64-bit numbers. Note that using explicit real type definitions should lead to more portable code.

The type Short_Float is not implemented.

Table 3-2. Attributes of Predefined Types Float and Long_Float

Attribute	Type	
	Float	Long_Float
'Machine_Overflows	TRUE	TRUE
'Machine_Rounds	TRUE	TRUE
'Machine_Radix	2	2
'Machine_Mantissa	24	53
'Machine_Emax	127	1023
'Machine_Emin	-125	-1021
'Mantissa	21	51
'Digits	6	15
'Size	32	64
'Emax	84	204
'Safe_Emax	125	1021
'Epsilon	9.53674E-07	8.88178E-16
'Safe_Large	4.25253E+37	2.24711641857789E+307
'Safe_Small	1.17549E-38	2.22507385850721E-308
'Large	1.93428E+25	2.57110087081438E+61
'Small	2.58494E-26	1.99469227433161E-62

3.3. LRM Chapter 4 - Names and Expressions

[LRM 4.10] Universal Expressions. There is no limit on the accuracy of real literal expressions. Real literal expressions are computed using an arbitrary-precision arithmetic package.

3.4. LRM Chapter 9 - Tasks

[LRM 9.6] Delay Statements, Duration, and Time. This implementation uses 32-bit fixed point numbers to represent the type Duration. The attributes of the type Duration are shown in Table 3-3.

Table 3-3. Attributes of Type Duration

Attribute	Value
'Delta	0.000122070E-03
'First	-86400.0
'Last	86400.0

[LRM 9.8] Priorities. Sixty-four levels of priority are available to associate with tasks through pragma Priority. The predefined subtype Priority is specified in the package System as

```
subtype Priority is Integer range 0..63;
```

Currently the priority assigned to tasks without a pragma Priority specification is 31; that is:

```
(System.Priority'First + System.Priority'Last) / 2
```

[LRM 9.11] Shared Variables. The restrictions on shared variables are only those specified in the LRM.

3.5. LRM Chapter 10 - Program Structure and Compilation Issues

[LRM 10.1] Compilation Units - Library Units. All main programs are assumed to be parameterless procedures or functions that return an integer result type.

3.6. LRM Chapter 11 - Exceptions

[LRM 11.1] Exception Declarations. Numeric_Error is raised for integer or floating point overflow and for divide-by-zero situations. Floating point underflow yields a result of zero without raising an exception.

Program_Error and Storage_Error are raised by those situations specified in LRM Section 11.1. Exception handling is also discussed in the Programming Guide chapter.

3.7. LRM Chapter 13 - Implementation-Dependent Features

As shown in Table 3-4, the current release of TeleGen2 supports most LRM Chapter 13 facilities. The sections below the table document those LRM Chapter 13 facilities that are either not implemented or that require explanation. Facilities implemented exactly as described in the LRM are not mentioned.

Table 3-4. Summary of LRM Chapter 13 Features for TeleGen2

13.1	Representation Clauses	Supported, except as indicated below (LRM 13.2 - 13.5). Pragma Pack is supported, *except for* dynamically sized components. For details on the TeleGen2 implementation of pragma Pack, see Section 3.7.1.
13.2	Length Clauses	Supported: 'Size 'Storage_Size for collections 'Storage_Size for task activation 'Small for fixed-point types See Section 3.7.2 for more information.
13.3	Enumeration Rep. Clauses	Supported, *except for* type Boolean or types derived from Boolean. (Note: users can easily define a non-Boolean enumeration type and assign a representation clause to it.)
13.4	Record Rep. Clauses	Supported *except for* records with dynamically sized components. See Section 3.7.4 for a full discussion of the TeleGen2 implementation.
13.5	Address Clauses	*Supported for:* objects (including task objects). *Not supported for:* packages, subprograms, or task units. See Section 3.7.5 for more information.
13.5.1	Interrupts	For interrupt entries, the address of a TeleGen2-defined *interrupt descriptor* can be given. See "Interrupt Handling" in the Programming Guide chapter for more information.
13.6	Change of Representation	Supported, *except for* types with record representation clauses.

------ *Continued on the next page* -----

Table 3–4. Summary of LRM Chapter 13 Features for TeleGen2 (Contd)

	—— Continued from the previous page ——	
13.7	Package System	Conforms closely to LRM model. Refer to Section 3.7.7 for details on the TeleGen2 implementation.
13.7.1	System-Dependent Named Numbers	Refer to the specification of package System (Section 3.7.7).
13.7.2	Representation Attributes	Implemented as described in LRM *except that*: 'Address for packages is unsupported. 'Address of a constant yields a null address.
13.7.3	Representation Attributes of Real Types	See Table 3-2.
13.8	Machine Code Insertions	Fully supported. The TeleGen2 implementation defines an attribute, 'Offset, that, along with the language-defined attribute 'Address, allows addresses of objects and offsets of data items to be specified in stack frames. Refer to "Using Machine Code Insertions" in the Programming Guide chapter for a full description on the implementation and use of machine code insertions.
13.9	Interface to Other Languages	Pragma Interface is supported for Assembly, C, UNIX and Fortran. Refer to "Interfacing to Other Languages" in the Programming Guide chapter for a description of the implementation and use of pragma Interface.
13.10	Unchecked Programming	Supported except as noted below (LRM 13.10.2).
13.10.1	Unchecked Storage Deallocation	Supported.
13.10.2	Unchecked Type Conversions	Supported *except for* unconstrained record or array types.

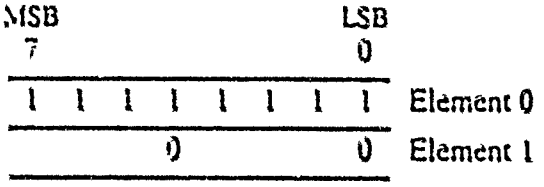
3.7.1. Pragma Pack.

This section discusses how pragma Pack is used in the TeleGen2 implementation.

a. With Boolean Arrays. You may pack Boolean arrays by the use of pragma Pack. The compiler allocates 8 bits for a single Boolean, 8 bits for a component of an unpacked Boolean array, and 1 bit for a component of a packed Boolean array. The first figure illustrates the layout of an unpacked Boolean array; the one below that illustrates a packed Boolean array:

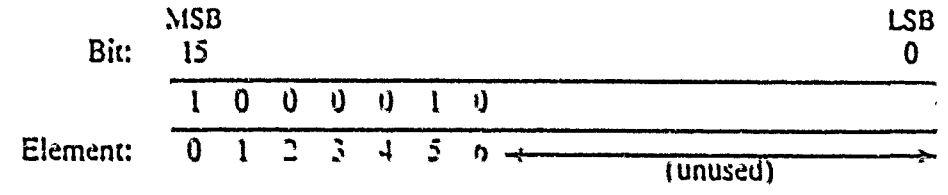
---------- Unpacked Boolean array: ---------

```
Unpacked_Bool_Arr_Type is array (Natural range 0..1) of Boolean
U_B_Arr: Unpacked_Bool_Arr_Type := (True,False);
```



---------- Packed Boolean array: ---------

```
Packed_Bool_Arr_Type is array (Natural range 0..6) of Boolean;
    pragma Pack (Packed_Bool_Arr_Type);
    P_B_Arr: Packed_Bool_Arr_Type := (P_B_Arr(1) => True,
      P_B_Arr(5) => True, others => False);
```



b. With Records. You may pack records by use of pragma Pack. Packed records follow these conventions:

1. The total size of the record is a multiple of 8 bits.
2. Packed records may cross word boundaries.
3. Records are packed to the bit level if the elements are themselves packed.

Below is an example of packing in a procedure, Rep_Proc, that defines three records of different lengths. Objects of these three packed record types are components of the packed record Rec. The storage allocated for Rec is 16 bits; that is, it is maximally packed.

```
procedure Rep_Proc is

  type A1 is array (Natural range 0 .. 8) of Boolean;
  pragma Pack (A1);

  type A2 is array (Natural range 0 .. 3) of Boolean;
  pragma Pack (A2);

  type A3 is array (Natural range 0 .. 2) of Boolean;
  pragma Pack (A3);

  type A_Rec is
    record
      One   : A1;
      Two   : A2;
      Three : A3;
    end record;
  pragma Pack (A_Rec);

  Rec   : A_Rec;

begin
  Rec.One       := ( 0 => True,   1 => False,  2 => False,
                     3 => False,  4 => True,   5 => False,
                     6 => False,  7 => False,  8 => True );
  Rec.Two (3)   := True;
  Rec.Three (1) := True;
end Rep_Proc;
```

3.7.2. [LRM 13.2] Length Clauses.

A length clause specifies an amount of storage associated with a type. The sections below describe how length clauses are supported in this implementation of TeleGen2 and how to use length clauses effectively within the context of TeleGen2.

3.7.2.1. (a) Specifying Size: T'Size.

The prefix T denotes an object. The size specification must allow for enough storage space to accommodate every allowable value of these objects. The constraint on the object (or on its subcomponents if any) must be static. For an unconstrained array type, the index subtypes must also be static.

For this implementation, Min_Size is the smallest number of bits logically required to hold any value in the range: no sign bit is allocated for non-negative ranges. Biased representations are not supported: e.g., a range of 100 .. 101 requires 7 bits, not 1. Warning: in the current release, using a size clause for a discrete type may cause inefficient code to be generated. For example, given...

```
type Nibble is range 0 .. 15;
for Nibble'Size use 4;
```

...each object of type Nibble will occupy only 4 bits, and relatively expensive bit-field instructions will be used for operations on Nibbles. (A single declared object of type Nibble will be aligned on a storage-unit boundary, however.)

For floating-point and access types, a size clause has no effect on the representation. (Task types are implemented as access types).

For composite (array or record) types, a size clause acts like an implicit pragma Pack, followed by a check that the resulting size is no greater than the requested size. Note that the composite type will be packed whether or not it is necessary to meet the requested size. The size clause for a record must be a multiple of storage units.

3.7.2.2. (b) Specifying Collection Size: T'Storage_Size.

A collection is the entire set of objects created by evaluation of allocators for an access type.

The prefix T denotes an access type. Given an access type Acc_Type, a length clause for a collection allocated using Acc_Type objects might look like this:

```
for Acc_Type'Storage_Size use 64;
```

In TeleGen2, the above length clause allocates from the heap 64 bytes of contiguous memory for objects created by Acc_Type allocators. Every time a new object is created, it is put into the remaining free part of the memory allocated for the collection, provided there is adequate space remaining in the collection. Otherwise, a storage error is raised.

Keeping the objects in a contiguous span of memory allows system storage reclamation routines to deallocate and manage the space when it is no longer needed. Pragma Controlled can prevent the deallocation of a specified collection of objects. Objects can be explicitly deallocated by calling the Unchecked_Deallocation procedure instantiated for the object and access types.

Given an access type which does not have a length clause specified, the 'Storage_Size attribute will return a value of 0.

Header Record

In this configuration of TeleGen2, information needed to manage storage blocks in a collection is stored in a collection header that requires 20 bytes of memory, adjacent to the collection, in addition to the value specified in the length clause.

Minimum Size

When an object is deallocated from a collection, a record containing link and size information for the space is put in the deallocated space as a placeholder. This enables the space to be located and reallocated. The space allocated for an object must therefore have the minimum size needed for the placeholder record. For this TeleGen2 configuration, this minimum size is the sum of the sizes of an access type and a integer type, or 6 bytes.

Dynamically Sized Objects

When a dynamically-sized object is allocated, a record requiring 2 bytes accompanies it to keep track of the size of the object for when it is put on the free list. The record is used to set the size field in the placeholder record since compaction may modify the value.

Size Expressions

Instead of specifying an integer in the length clause, you can use an expression to specify storage for a given number of objects. For example, suppose an access type Dict_Ref references a record Symbol_Rec containing five fields:

```
type Tag is String(1..8);

type Symbol_Rec;
type Dict_Ref is access Symbol_Rec;

type Symbol_Rec is
  record
    Left   : Dict_Ref;
    Right  : Dict_Ref;
    Parent : Dict_Ref;
    Value  : Integer;
    Key    : Tag;
   end record;
```

To allocate 10 Symbol_Rec objects, you could use an expression such as:

```
for Dict_Ref'Storage_Size use ((Symbol_Rec'Size * 10)+20);
```

where 20 is the extra space needed for the header record. (Symbol_Rec is obviously larger than the minimum size required, which is equivalent to one access type and one integer.)

In another implementation, Symbol_Rec might be a variant record that uses a variable length for the string Key:

```
type Symbol_Rec(Last : Natural :=0) is
  record
    Left   : Dict_Ref;
```

```
    Right  : Dict_Ref;
    Parent : Dict_Ref;
    Value  : Integer;
    Key    : String(1..Last);
  end record;
```

In this case, Symbol_Rec objects would be dynamically sized depending on the length of the string for Key. Using a length clause for Dict_Ref as above would then be illegal since Symbol_Rec'Size cannot be consistently determined. A length clause for Symbol_Rec objects, as described in (a) above, would be illegal since not all components of Symbol_Rec are static. As defined, a Symbol_Rec object could conceivably have a Key string with Integer'Last number of characters.

3.7.2.3. (c) Specifying Storage for Task Activation: T'Storage_Size.

The prefix T denotes a task type. A length clause for a task type specifies the number of storage units to be reserved for an activation of a task of the type. The TeleGen2 default stack size is 4000 bytes.

3.7.2.4. (d) Specifying 'Small for Fixed Point Types: T'Small.

Small is the absolute precision (a positive real number) while the prefix T denotes the first named subtype of a fixed point type. Elaboration of a real type defines a set of model numbers. T'Small is generally a power of 2, and model numbers are generally multiples of this number so that they can be represented exactly on a binary machine. All other real values are defined in terms of model numbers having explicit error bounds.

Example:

```
type Fixed is delta 0.25 range -10.0 .. 10.0;
```

Here...

Fixed'Small = 0.25 -- *A power of 2*

3.0 = 12 * 0.25 -- *A model number but not a power of 2*

The value of the expression of the length clause must not be greater than the delta of the first named subtype. The effect of the length clause is to use this value of 'Small for the representation of values of the fixed point base type. The length clause thereby also affects the amount of storage for objects that have this type.

If a length clause is not used, for model numbers defined by a fixed point constraint, the value of Small is defined as the largest power of two that is not greater than the delta of the fixed accuracy definition.

If a length clause is used, the model numbers are multiples of the specified value for Small. For this configuration of TeleGen2, the specified value must be (mathematically) equal to either an exact integer or the reciprocal of an exact integer.

Examples:

```
1.0, 2.0, 3.0, 4.0, . . . are legal
0.5, 1.0/3.0, 0.25, 1.0/3600.0 are legal
2.5, 2.0/3.0, 0.3 are illegal
```

3.7.3. [LRM 13.3] Enumeration Representation Clauses.

Enumeration representation clauses are supported, except for Boolean types.

Performance note: Be aware that use of such clauses will introduce considerable overhead into many operations that involve the associated type. Such operations include indexing an array by an element of the type, or computing the 'Pos, 'Pred, or 'Succ attributes for values of the type.

3.7.4. [LRM 13.4] Record Representation Clauses.

Since record components are subject to rearrangement by the compiler, you must use representation clauses to guarantee a particular layout. Such clauses are subject to the following constraints:

- Each component of the record must be specified with a component clause.
- The alignment of the record is restricted to mods 1, 2, and 4; byte, word, and long-word aligned, respectively.
- Bits are ordered right to left within a byte.
- Components may cross word boundaries.

Here is a simple example showing how the layout of a record can be specified by using representation clauses:

```
package Repspec_Example is
  Bits : constant := 1;
  Word : constant := 4;

  type Five is range 0 .. 16#1F#;
  type Seventeen is range 0 .. 16#1FFFF#;
  type Nine is range 0 .. 511;

  type Record_Layout_Type is record
      Element1 : Seventeen;
      Element2 : Five;
      Element3 : Boolean;
      Element4 : Nine;
    end record;

  for Record_Layout_Type use record at mod 2;
      Element1 at 0*Word range 0 .. 16;
      Element2 at 0*Word range 17 .. 21;
```

```
      Element3 at 0*Word range 22 .. 22;
      Element4 at 0*Word range 23 .. 31;
    end record;

  Record_Layout : Record_Layout_Type;
end Repspec_Example;
```

3.7.5. [LRM 13.5] Address Clauses.

The Ada compiler supports address clauses for objects and entries. Address clauses for packages and task units are not supported.

Address clauses for objects may be used to access hardware memory registers or other known memory locations. The use of address clauses is affected by the fact that the System.Address type is private. For the MC68k0 target, literal addresses are represented as integers, so an unchecked conversion must be applied to these literals before they can be passed as parameters of type System.Address. For example, in the examples in this document the following declaration is often assumed:

```
function Addr is new Unchecked_Conversion (Long_Integer,
                                           System.Address);
```

This function is invoked when an address literal needs to be converted to an Address type. Naturally, user programs may implement a different convention. Below is a sample program that uses address clauses and this convention. Package System must be explicitly *with*ed when using address clauses.

```
with System;
with Unchecked_Conversion;
procedure Hardware_Access is
  function Addr is new Unchecked_Conversion (Long_Integer,
                                             System.Address);
  Hardware_Register : Integer;
  for Hardware_Register use at Addr (16#FF0000#);
begin
  ...
end Hardware_Access;
```

When using an address clause for an object with an initial value, the address clause should immediately follow the object declaration:

```
Obj: Some_Type := <init_expr>;
for Obj use at <addr_expr>;
```

This sequence allows the compiler to perform an optimization wherein it generates code to evaluate the <addr_expr> as part of the elaboration of the declaration of the object. The expression <init_expr> will then be evaluated and assigned directly to the object, which is stored at <addr_expr>. If another declaration had intervened between the object declaration and the address clause, the compiler would have had to create a temporary object to hold the initialization value before copying it into the object when the address clause is elaborated. If

the object were a large composite type, the need to use a temporary could result in considerable overhead in both time and space. To optimize your applications, therefore, you are encouraged to place address clauses immediately after the relevant object declaration.

As mentioned above, arrays containing components that can be allocated in a signed or unsigned byte (8 bits) are packed, one component per byte. Furthermore, such components are referenced in generated code by MC68S0x0 byte instructions. The following example indicates how these facts allow access to hardware byte registers:

```
with System;
with Unchecked_Conversion;
procedure Main is
  function Addr is new Unchecked_Conversion (Long_Integer,
                                             System.Address);
  type Byte is range -128..127;
  HW_Regs : array (0..1) of Byte;
  for HW_Regs use at Addr (16#FFF310#);

  Status_Byte : constant Integer := 0;
  Next_Block_Request: constant Integer := 1;
  Request_Byte : Byte := 119;
  Status : Byte;

begin
  Status := HW_Regs(Status_Byte);
  HW_Regs(Next_Block_Request) := Request_Byte;
end Main;
```

Two byte hardware registers are referenced in the example above. The status byte is at location 16#FFF310# and the next block request byte is at location 16#FFF311#.

Function Addr takes a long integer as its argument. Long_Integer'Last is 16#7FFFFFFF#, but there are certainly addresses greater than Long_Integer'Last. Those addresses with the high bit set, such as FFFA0000, cannot be represented as a positive long integer. Thus, for addresses with the high bit set, the address should be computed as the negation of the 2's complement of the desired address. According to this method, the correct representation of the sample address above would be Addr(-16#00060000#).

3.7.6. [LRM 13.6] Change of Representation.

TeleGen2 supports changes of representation, except for types with record representation clauses.

3.7.7. [LRM 13.7] The Package System.

The specification of TeleGen2's implementation of package System is presented in the LRM Appendix F section at the end of this chapter.

3.7.8. [LRM 13.7.2] Representation Attributes.

The compiler does not support 'Address for packages.

3.7.9. [LRM 13.7.3] Representation Attributes of Real Types.

The representation attributes for the predefined floating point types were presented in Table 3-2.

3.7.10. [LRM 13.8] Machine Code Insertions.

Machine code insertions, an optional feature of the Ada language, are fully supported in TeleGen2. Refer to the "Using Machine Code Insertions" section in the Programming Guide chapter for information regarding their implementation and for examples on their use.

3.7.11. [LRM 13.9] Interface to Other Languages.

In TeleGen2, pragma Interface is supported for Assembly, C, UNIX, and Fortran. Refer to "Interfacing to Other Languages" in the Programming Guide chapter for for information on the use of pragma Interface. TeleGen2 does not currently allow pragma Interface for library units.

3.7.12. [LRM 13.10] Unchecked Programming.

Unchecked_Conversion is allowed except when the target data subtype is an unconstrained array or record type. If the size of the source and target are static and equal, the compiler will perform a bitwise copy of data from the source object to the target object.

Where the sizes of source and target differ, the following rules will apply.

- If the size of the source is greater than the size of the target, the high address bits will be truncated in the conversion.
- If the size of the source is less than the size of the target, the source will be moved into the low address bits of the target.

The compiler will issue a warning when Unchecked_Conversion is instantiated with unequal sizes for source and target subtype. Unchecked_Conversion between objects of different or non-static sizes will usually produce less efficient code and should be avoided, if possible.

3.8. LRM Appendix F for TeleGen2

The Ada language definition allows for certain target dependencies. These dependencies must be described in the reference manual for each implementation, in an "Appendix F" that addresses each point listed in LRM Appendix F. Table 3-5 constitutes Appendix F for this implementation. Points that require further clarification are addressed in sections referenced in the table.

Table 3-5. LRM Appendix F for TeleGen2

(1) Implementation-Dependent Pragmas	(a) Implementation-defined pragmas: Comment, Images, Interrupt, Interface_Information, Linkname, No_Suppress, Preserve_Layout, and Suppress_All (Section 3.8.1). (b) Predefined pragmas with implementation-dependent characteristics: • Interface (assembly, UNIX C, and Fortran-- see "Interfacing to Other Languages." Not supported for library units. • List and Page (in context of source/error compiler listings.) (See the *User Guide*.) • Pack. See Section 3.7.1. • Inline. Not supported for library-level subprograms. • Priority. Not supported for main programs. *Other supported predefined pragmas:* Controlled Shared Suppress Elaborate *Predefined pragmas partly supported* (see Section 3.1): Memory_Size Storage_Unit System_Name *Not supported:* Optimize

------- *Continued on the next page* -------

Table 3-5. LRM Appendix F for TeleGen2 (Contd)

—— *Continued from the previous page* ——

(2) Implementation-Dependent Attributes	'Offset. Used for machine code insertions. The predefined attribute 'Address is not supported for packages. See "Using Machine Code Insertions" earlier in this chapter for information on 'Offset and 'Address. 'Extended_Image 'Extended_Value 'Extended_Width 'Extended_Aft 'Extended_Digits 'Extended_Fore Refer to Section 3.8.2 for information on the implementation-defined extended attributes listed above.
(3) Package System	See Section 3.7.7.
(4) Restrictions on Representation Clauses	Summarized in Table 3-4.
(5) Implementation-Generated Names	None
(6) Address Clause Expression Interpretation	An expression that appears in an object address clause is interpreted as the address of the first storage unit of the object.
(7) Restrictions on Unchecked Conversions	Summarized in Table 3-4.

------- *Continued on the next page* -------

Table 3-5. LRM Appendix F for TeleGen2 (Contd)

—— *Continued from the previous page* ——

(8) Implementation-Dependent Characteristics of the I/O Packages.	1. In Text_IO, the type Count is defined as follows:
	`type Count is range 0..(2 ** 31)-2`
	2. In Text_IO, the type Field is defined as follows:
	`subtype Field is integer range 0..1000`
	3. In Text_IO, the Form parameter of procedures Create and Open is not supported. (If you supply a Form parameter with either procedure, it is ignored.)
	4. Sequential_IO and Direct_IO cannot be instantiated for unconstrained array types or discriminated types without defaults.
	5. The standard library contains preinstantiated versions of Text_IO.Integer_IO for types Integer and Long_Integer and of Text_IO.Float_IO for types Float and Long_Float. We suggest that you use the following to eliminate multiple instantiations of these packages:
	`Short_Integer_Text_IO`
	`Integer_Text_IO`
	`Long_Integer_Text_IO`
	`Float_Text_IO`
	`Long_Float_Text_IO`

3.8.1. Implementation-Defined Pragmas.

There are eight implementation-defined pragmas in TeleGen2: pragmas Comment, Images, Interface_Information, Interrupt, Linkname, No_Suppress, Preserve_Layout, and Suppress_All.

3.8.1.1. Pragma Comment.

Pragma Comment is used for embedding a comment into the object code. Its syntax is:

```
pragma Comment ( <string_literal> );
```

where "<string_literal>" represents the characters to be embedded in the object code. Pragma Comment is allowed only within a declarative part or immediately within a package specification. Any number of comments may be entered into the object code by use of pragma Comment.

3.8.1.2. Pragma Images.

Pragma Images controls the creation and allocation of the image and index tables for a specified enumeration type. The image table is a literal string consisting of enumeration literals catenated together. The index table is an array of integers specifying the location of each literal within the image table. The length of the index table is therefore the sum of the lengths of the literals of the enumeration type; the length of the index table is one greater than the number of literals.

The syntax of this pragma is:

```
pragma Images( <enumeration_type>, Deferred);
    -- or --
pragma Images( <enumeration_type>, Immediate);
```

The default, Deferred, saves space in the literal pool by not creating image and index tables for an enumeration type unless the 'Image, 'Value, or 'Width attribute for the type is used. If one of these attributes is used, the tables are generated in the literal pool of the compilation unit in which the attribute appears. If the attributes are used in more than one compilation unit, more than one set of tables is generated, eliminating the benefits of deferring the table. In this case, using

```
pragma Images( <enumeration_type>, Immediate);
```

will cause a single image table to be generated in the literal pool of the unit declaring the enumeration type.

For a very large enumeration type, the length of the image table will exceed Integer'Last (the maximum length of a string). In this case, using either

```
pragma Images( <enumeration_type>, Immediate);
```

or the 'Image, 'Value, or 'Width attribute for the type will result in an error message from the compiler.

3.8.1.3. Pragma Interface_Information.

The existing Ada interface pragma only allows specification of a language name. In some cases, the optimizing code generator will need more information than can

be derived from the language name. Therefore there is a need for an implementation-specific pragma, Interface_Information.

There is an extended usage of this pragma for Machine Code Insertion procedures which does not use a preceding pragma Interface. Other than that case, a pragma Interface_Information is always associated with a Pragma Interface. The syntax is:

```
pragma Interface_Information (Name,
                              Link_Name,
                              Mechanism,
                              Parameters,
                              Clobbered_Regs);
```

where:

```
name                :== ada_subprogram_identifier, required
link_name           :== string, default = ""
mechanism           :== string, default = "PROTECTED"
parameters          :== string, default = ""
clobbered_regs      :== string, default = ""
```

Scope.

Pragma Interface_Information is allowed wherever the standard pragma Interface is allowed, and must be immediately preceded by a pragma Interface purporting to the same Ada subprogram, in the same declarative part or package specification; no intervening declaration is allowed between the Interface and Interface_Information pragmas. Contrary to pragma Interface, this pragma is not allowed for overloaded subprograms (it specifies information that pertain to one specific body of non-Ada code). If the user wishes to use overloaded Ada names, the Interface and Interface_Information pragmas may be applied to unique renaming declarations.

The pragma is also allowed for a library unit; in that case, the pragma must occur immediately after the corresponding Interface pragma, and before any subsequent compilation unit.

This pragma may be applied to any interfaced subprogram, regardless of the language or system named in the interface pragma. The code generator is responsible for rejecting or ignoring illegal or redundant interface information. The optimizing code generator will process and check the legality of such interfaced subprograms at the time of the spec compilation, instead of waiting for an actual use of the interfaced subprogram. This will save the user from extensive recompilation of the offensive specification and all its dependents should an illegal pragma have been used.

This pragma is also used for Machine Code Insertion (MCI) procedures. In that case, the "mechanism" should be set to "mci." This allows the user to specify detailed parameter characteristics for the call and inlined call to the MCI procedure.

When used in conjunction with pragma Inline, this allows the user to directly insert a minimal set of instructions into the call location.

Parameters.

Name: Ada subprogram identifier. The rule detailed in LRM 13.9 for a subprogram named in a pragma Interface apply here as well. As explained above, the subprogram must have been named in an immediately preceding Interface pragma.

This is the only required parameter. Since the other parameters are optional, positional association may only be used if all parameters are specified, or only the rightmost ones are defaulted. Named association must be used otherwise: this requires that the front-end supports it in pragmas.

Link_Name: string literal. When specified, this parameter indicates the name the code generator must use to reference the named subprogram. This string name may contain any characters allowed in an Ada string and must be passed unchanged (in particular, not case-mapped) to the code generator. The code generator will reject names that are illegal in the particular language or system being targeted.

If this parameter is not specified, it defaults to a null string. The code generator will interpret a default link_name differently, depending on the target language/system (the default is generally the Ada name, or is derived from it, for example, "_Ada_name" for 'C' calls).

Mechanism: string literal. The only mechanism currently implemented is the "mci" mechanism used strictly in conjunction with Machine Code Insertion procedures.

A future mechanism will include "protected" where the code generator will protect the Ada exception model on calls to that external subprogram. Unfortunately, such protection is costly, which means that it should be applied only when necessary, as most interfaced subprograms do not raise exceptions or traps. Until this mechanism is implemented, external code which causes traps will cause unpredictable unhandled exception tracebacks.

Parameters: string literal. This string, when present, tells the code generator that some parameters are to be passed in registers (instead of on the stack, which is the default). The string is passed as is by the front-end, and is decoded by the code generator.

The code generator interprets the string as a positional aggregate, where each position refers to a parameter of the interfaced subprogram. Each position of the aggregate may either be null, or one of the following identifiers: "D0, D1, A0, A1, FP0, or FP1," which specify that the corresponding parameter be passed in the named register. Thus the string "A0,,D0" specifies that the first parameter be passed in A0, the second in the stack, the third in D0, and any other parameters in

the stack.

A register identifier may be used, at most, once in the parameter passing description string, except for function return value, as explained below. The code generator will also enforce some semantic restrictions; for example, a floating point register can only be used to pass a float type parameter, although a float parameter can be passed in a regular data register. Only scalar values and addresses may be passed directly in registers; composites must be passed in the stack (by reference or by value, dictated by the code generator model for the target language/system). Explicit passing of the 'Address of a composite in a register is allowed (the parameter must be of type System.Address).

The parameters string can also be used to specify the register in which a function return value will be passed back. The parameters registers string may in that case have one position more than the number of parameters; this position describes the return value. The register used for the return value may have been used for an input parameter. This is the only case where a register may occur twice in the same string. If not specified for a given function, the code generator uses the register normally used for the target language/system (that is, "C" returns all in D0 or D0/D1, whereas "Assembly" returns in D0, D0/D1, A0, or FP0 depending on the return value type).

Clobbered_Regs. These are the registers (comma-separated list) that are destroyed by this operation. The code generator will keep anything valuable in these registers at the point of the call.

A simple example of the use of pragma Interface_Information is:

```
procedure Do_Something (Addr: System.Address; Len: Integer);
pragma Interface (Assembly, Do_Something);
pragma Interface_Information ( Name       => Do_Something,
                               Link_Name  => "CGS$$DOIT",
                               Mechanism  => "UNPROTECTED",
                               Parameters => "A0,D0");
```

3.8.1.4. Pragma Interrupt.

Pragma Interrupt is described in this reference volume in Chapter 2. Please refer to the section "Optimized Interrupt Entries," subsection "Function-Mapped Optimizations."

3.8.1.5. Pragma Linkname.

Pragma Linkname is used to provide interface to any routine whose name can be specified by an Ada string literal. This allows access to routines whose identifiers do not conform to Ada identifier rules.

Pragma Linkname takes two arguments. The first is a subprogram name that has been previously specified in a pragma Interface statement. The second is a string literal specifying the exact link name to be employed by the code generator in emitting calls to the associated subprogram. The syntax is:

```
pragma Interface ( assembly, <subprogram_name> );
pragma Linkname ( <subprogram_name>, <string_literal> );
```

If pragma Linkname does not immediately follow the pragma Interface for the associated program, a warning will be issued saying that the pragma has no effect.

A simple example of the use of pragma Linkname is:

```
procedure Dummy_Access( Dummy_Arg : System.Address );
pragma Interface (assembly, Dummy_Access );
pragma Linkname (Dummy_Access, "_access");
```

Notes: (1) It is preferable that the user use pragma Interface_Information for this functionality.

(2) In the example above, the link name generated may be "__access" instead of "_access" if your native C compiler prepends an underscore to the specified link name.

3.8.1.6. Pragma No_Suppress.

No_Suppress is a TeleGen2-defined pragma that prevents the suppression of checks within a particular scope. It can be used to override pragma Suppress in an enclosing scope. No_Suppress is particularly useful when you have a section of code that relies upon predefined checks to execute correctly, but you need to suppress checks in the rest of the compilation unit for performance reasons.

Pragma No_Suppress has the same syntax as pragma Suppress and may occur in the same places in the source. The syntax is:

```
pragma No_Suppress (<identifier> [, [ON =>] <name>]);
```

where <identifier> is the type of check you do not want to suppress (e.g., access_check: refer to LRM 11.7)

<name> is the name of the object, type/subtype, task unit, generic unit, or subprogram within which the check is not to be suppressed: <name> is optional.

If neither Suppress nor No_Suppress is present in a program, checks will not be suppressed. You may override this default at the command level, by compiling the file with the -inhibit option and specifying with that option the type of checks you want to suppress. For more information on -inhibit, refer to your TeleGen2 *Overview and Command Summary* document.

If either Suppress or No_Suppress are present, the compiler uses the pragma that applies to the specific check in order to determine whether that check is to be made. If both Suppress and No_Suppress are present in the same scope, the pragma declared last takes precedence. The presence of pragma Suppress or No_Suppress in the source takes precedence over an -inhibit option provided during compilation.

3.8.1.7. Pragma Preserve_Layout.

The TeleGen2 compiler reorders record components to minimize gaps within records. Pragma Preserve_Layout forces the compiler to maintain the Ada source order of components of a given record type, thereby preventing the compiler from performing this record layout optimization.

The syntax of this pragma is:

Pragma Preserve_Layout (ON => Record_Type_Name)

Preserve_Layout must appear before any forcing occurrences of the record type and must be in the same declarative part, package specification, or task specification. This pragma can be applied to a record type that has been packed. If Preserve_Layout is applied to a record type that has a record representation clause, the pragma only applies to the components that do not have component clauses. These components will appear in Ada source order after the components with component clauses.

3.8.1.8. Pragma Suppress_All.

Suppress_All is a TeleGen2-defined pragma that will suppress all checks in a given scope. Pragma Suppress_All contains no arguments and can be placed in the same scopes as pragma Suppress.

In the absence of pragma Suppress_All or any other suppress pragma, the scope which contains the pragma will have checking turned off. This pragma should be used in a safe piece of time critical code to allow for better performance.

3.8.2. Implementation-Dependent Attributes.

3.8.2.1. 'Address and 'Offset.

These were discussed within the context of using machine code insertions, in the Programming Guide chapter.

3.8.2.2. Extended Attributes for Scalar Types.

The extended attributes extend the concept behind the Text_IO attributes 'Image, 'Value, and 'Width to give the user more power and flexibility when displaying values of scalars. Extended attributes differ in two respects from their predefined counterparts:

1. Extended attributes take more parameters and allow control of the format of the output string.

2. Extended attributes are defined for all scalar types, including fixed and floating point types.

Extended versions of predefined attributes are provided for integer, enumeration, floating point, and fixed point types:

Integer:	'Extended_Image,	'Extended_Value,	'Extended_Width
Enumeration:	'Extended_Image,	'Extended_Value,	'Extended_Width
Floating Point:	'Extended_Image,	'Extended_Value,	'Extended_Digits
Fixed Point:	'Extended_Image, 'Extended_Aft	'Extended_Value,	'Extended_Fore,

The extended attributes can be used without the overhead of including Text_IO in the linked program. Below is an example that illustrates the difference between instantiating Text_IO.Float_IO to convert a float value to a string and using Float'Extended_Image:

```
with Text_IO;
function Convert_To_String ( Fl : Float ) return String is

Temp_Str : String ( 1 .. 6 + Float'Digits );

package Flt_IO is new Text_IO.Float_IO (Float);

begin

  Flt_IO.Put ( Temp_Str, Fl );
  return Temp_Str;

end Convert_To_String;

    function Convert_To_String_No_Text_IO( Fl : Float ) return String is

    begin

      return Float'Extended_Image ( Fl );

    end Convert_To_String_No_Text_IO;
```

```
with Text_IO, Convert_To_String, Convert_To_String_No_Text_IO;
procedure Show_Different_Conversions is

Value : Float := 10.03376;

begin

  Text_IO.Put_Line ( "Using the Convert_To_String, the value of the variable
  is : " & Convert_To_String ( Value ) );
  Text_IO.Put_Line ( "Using the Convert_To_String_No_Text_IO, the value
  is : " & Convert_To_String_No_Text_IO ( Value ) );

end Show_Different_Conversions;
```

3.8.2.2.1. Integer Attributes

'Extended_Image

Usage:

X'Extended_Image(Item,Width,Base,Based,Space_IF_Positive)

Returns the image associated with Item as defined in Text_IO.Integer_IO. The Text_IO definition states that the value of Item is an integer literal with no underlines, no exponent, no leading zeros (but a single zero for the zero value), and a minus sign if negative. If the resulting sequence of characters to be output has fewer than Width characters, leading spaces are first output to make up the difference. (LRM 14.3.7:10,14.3.7:11)

For a prefix X that is a discrete type or subtype; this attribute is a function that may have more than one parameter. The parameter Item must be an integer value. The resulting string is without underlines, leading zeros, or trailing spaces.

Parameter Descriptions:

Item	The item for which you want the image; it is passed to the function. *Required*
Width	The minimum number of characters to be in the string that is returned. If no width is specified, the default (0) is assumed. *Optional*
Base	The base in which the image is to be displayed. If no base is specified, the default (10) is assumed. *Optional*
Based	An indication of whether you want the string returned to be in base notation or not. If no preference is specified, the default (false) is assumed. *Optional*
Space_If_Positive	An indication of whether or not the sign bit of a positive integer is included in the string returned. If no preference is specified, the default (false) is assumed. *Optional*

Examples:

Suppose the following subtype were declared:

```
subtype X is Integer Range -10..16;
```

Then the following would be true:

```
X'Extended_Image(5)                    = "5"
X'Extended_Image(5,0)                  = "5"
X'Extended_Image(5,2)                  = " 5"
X'Extended_Image(5,0,2)                = "101"
X'Extended_Image(5,4,2)                = " 101"
X'Extended_Image(5,0,2,True)           = "2#101#"
X'Extended_Image(5,0,10,False)         = "5"
X'Extended_Image(5,0,10,False,True)    = " 5"
X'Extended_Image(-1,0,10,False,False)  = "-1"
X'Extended_Image(-1,0,10,False,True)   = "-1"
X'Extended_Image(-1,1,10,False,True)   = "-1"
X'Extended_Image(-1,0,2,True,True)     = "-2#1#"
X'Extended_Image(-1,10,2,True,True)    = "      -2#1#"
```

'Extended_Value

Usage:

X'Extended_Value(Item)

Returns the value associated with Item as defined in Text_IO.Integer_IO. The Text_IO definition states that given a string, it reads an integer value from the be-

ginning of the string. The value returned corresponds to the sequence input. (LRM 14.3.7:14)

For a prefix X that is a discrete type or subtype, this attribute is a function with a single parameter. The actual parameter Item must be of predefined type string. Any leading or trailing spaces in the string X are ignored. In the case where an illegal string is passed, a Constraint_Error is raised.

Parameter Description:

Item	A parameter of the predefined type string; it is passed to the function. The type of the returned value is the base type X. *Required*

Examples:

Suppose the following subtype were declared:

```
Subtype X is Integer Range -10..16;
```

Then the following would be true:

```
X'Extended_Value("5")          = 5
X'Extended_Value(" 5")         = 5
X'Extended_Value("2#101#")     = 5
X'Extended_Value("-1")         = -1
X'Extended_Value(" -1")        = -1
```

'Extended_Width

Usage:

X'Extended_Width(Base,Based,Space_If_Positive)

Returns the width for subtype of X.

For a prefix X that is a discrete subtype: this attribute is a function that may have multiple parameters. This attribute yields the maximum image length over all values of the type or subtype X.

Parameter Descriptions:

Base	The base for which the width will be calculated. If no base is specified, the default (10) is assumed. *Optional*
Based	An indication of whether the subtype is stated in based notation. If no value for based is specified, the default (false) is assumed. *Optional*
Space_If_Positive	An indication of whether or not the sign bit of a positive integer is included in the string returned. If no preference is specified, the default (false) is assumed. *Optional*

Examples:

Suppose the following subtype were declared:

```
Subtype X is Integer Range -10..16;
```

Then the following would be true:

```
X'Extended_Width                      = 3  - "-10"
X'Extended_Width(10)                  = 3  - "-10"
X'Extended_Width(2)                   = 5  - "10000"
X'Extended_Width(10,True)             = 7  - "-10#10#"
X'Extended_Width(2,True)              = 8  - "2#10000#"
X'Extended_Width(10,False,True)       = 3  - " 16"
X'Extended_Width(10,True,False)       = 7  - "-10#10#"
X'Extended_Width(10,True,True)        = 7  - " 10#16#"
X'Extended_Width(2,True,True)         = 9  - " 2#10000#"
X'Extended_Width(2,False,True)        = 6  - " 10000"
```

3.8.2.2.2. Enumeration Type Attributes

'Extended_Image

Usage:

X'Extended_Image(Item,Width,Uppercase)

Returns the image associated with Item as defined in Text_IO.Enumeration_IO. The Text_IO definition states that given an enumeration literal, it will output the value of the enumeration literal (either an identifier or a character literal). The character case parameter is ignored for character literals. (LRM 14.3.9:9)

For a prefix X that is a discrete type or subtype; this attribute is a function that may have more that one parameter. The parameter Item must be an enumeration value. The image of an enumeration value is the corresponding identifier, which may have character case and return string width specified.

Parameter Descriptions:

Item	The item for which you want the image: it is passed to the function. *Required*
Width	The minimum number of characters to be in the string that is returned. If no width is specified, the default (0) is assumed. If the Width specified is larger than the image of Item, the return string is padded with trailing spaces. If the Width specified is smaller than the image of Item, the default is assumed and the image of the enumeration value is output completely. *Optional*
Uppercase	An indication of whether the returned string is in uppercase characters. In the case of an enumeration type where the enumeration literals are character literals, Uppercase is ignored and the case specified by the type definition is taken. If no preference is specified, the default (true) is assumed. *Optional*

Examples:

Suppose the following types were declared:

```
type X is (red, green, blue, purple);
type Y is ('a', 'B', 'c', 'D');
```

Then the following would be true:

```
X'Extended_Image(red)              = "RED"
X'Extended_Image(red, 4)           = "RED "
X'Extended_Image(red,2)            = "RED"
X'Extended_Image(red,0,false)      = "red"
X'Extended_Image(red,10,false)     = "red       "
Y'Extended_Image('a')              = "'a'"
Y'Extended_Image('B')              = "'B'"
Y'Extended_Image('a',6)            = "'a'  "
Y'Extended_Image('a',0,true)       = "'a'"
```

'Extended_Value

Usage:

X'Extended_Value(Item)

Returns the image associated with Item as defined in Text_IO.Enumeration_IO. The Text_IO definition states that it reads an enumeration value from the beginning of the given string and returns the value of the enumeration literal that corresponds to the sequence input. (LRM 14.3.9:11)

For a prefix X that is a discrete type or subtype; this attribute is a function with a single parameter. The actual parameter Item must be of predefined type string. Any leading or trailing spaces in the string X are ignored. In the case where an illegal string is passed, a Constraint_Error is raised.

Parameter Descriptions:

Item	A parameter of the predefined type string; it is passed to the function. The type of the returned value is the base type of X. *Required*

Examples:

Suppose the following type were declared:

```
type X is (red, green, blue, purple);
```

Then the following would be true:

```
X'Extended_Value("red")          = red
X'Extended_Value(" green")       = green
X'Extended_Value("    Purple")   = purple
X'Extended_Value(" GreEn  ")     = green
```

'Extended_Width

Usage:

X'Extended_Width

Returns the width for subtype of X.

For a prefix X that is a discrete type or subtype; this attribute is a function. This attribute yields the maximum image length over all values of the enumeration type or subtype X.

Parameter Descriptions:

There are no parameters to this function. This function returns the width of the largest (width) enumeration literal in the enumeration type specified by X.

Examples:

Suppose the following types were declared:

```
type X is (red, green, blue, purple);
type Z is (X1, X12, X123, X1234);
```

Then the following would be true:

```
X'Extended_Width     = 6  -- "purple"
Z'Extended_Width     = 5  -- "X1234"
```

3.8.2.2.3. Floating Point Attributes

'Extended_Image

Usage:

X'Extended_Image(Item,Fore,Aft,Exp,Base,Based)

Returns the image associated with Item as defined in Text_IO.Float_IO. The Text_IO definition states that it outputs the value of the parameter Item as a decimal literal with the format defined by the other parameters. If the value is negative, a minus sign is included in the integer part of the value of Item. If Exp is 0, the integer part of the output has as many digits as are needed to represent the integer part of the value of Item or is zero if the value of Item has no integer part. (LRM 14.3.8:13, 14.3.8:15)

Item must be a Real value. The resulting string is without underlines or trailing spaces.

Parameter Descriptions:

Item	The item for which you want the image: it is passed to the function. *Required*
Fore	The minimum number of characters for the integer part of the decimal representation in the return string. This includes a minus sign if the value is negative and the base with the '#' if based notation is specified. If the integer part to be output has fewer characters than specified by Fore, leading spaces are output first to make up the difference. If no Fore is specified, the default value (2) is assumed. *Optional*
Aft	The minimum number of decimal digits after the decimal point to accommodate the precision desired. If the delta of the type or subtype is greater than 0.1, then Aft is 1. If no Aft is specified, the default (X'Digits-1) is assumed. If based notation is specified, the trailing '#' is included in Aft. *Optional*
Exp	The minimum number of digits in the exponent. The exponent consists of a sign and the exponent, possibly with leading zeros. If no Exp is specified, the default (3) is assumed. If Exp is 0, no exponent is used. *Optional*
Base	The base that the image is to be displayed in. If no base is specified, the default (10) is assumed. *Optional*
Based	An indication of whether you want the string returned to be in based notation or not. If no preference is specified, the default (false) is assumed. *Optional*

Examples:

Suppose the following type were declared:

```
type X is digits 5 range -10.0 .. 16.0;
```

Then the following would be true:

```
X'Extended_Image(5.0)                    = " 5.0000E+00"
X'Extended_Image(5.0,1)                  = "5.0000E+00"
X'Extended_Image(-5.0,1)                 = "-5.0000E+00"
X'Extended_Image(5.0,2,0)                = " 5.0E+00"
X'Extended_Image(5.0,2,0,0)              = " 5.0"
X'Extended_Image(5.0,2,0,0,2)            = "101.0"
X'Extended_Image(5.0,2,0,0,2,True)       = "2#101.0#"
X'Extended_Image(5.0,2,2,3,2,True)       = "2#1.1#E+02"
```

'Extended_Value

Usage:

X'Extended_Value(Item)

Returns the value associated with Item as defined in Text_IO.Float_IO. The Text_IO definition states that it skips any leading zeros, then reads a plus or minus sign if present then reads the string according to the syntax of a real literal. The return value is that which corresponds to the sequence input. (LRM 14.3.8:9, 14.3.8:10)

For a prefix X that is a discrete type or subtype; this attribute is a function with a single parameter. The actual parameter Item must be of predefined type string. Any leading or trailing spaces in the string X are ignored. In the case where an illegal string is passed, a Constraint_Error is raised.

Parameter Descriptions:

Item	A parameter of the predefined type string; it is passed to the function. The type of the returned value is the base type of the input string. *Required*

Examples:

Suppose the following type were declared:

```
type X is digits 5 range -10.0 .. 16.0;
```

Then the following would be true:

```
X'Extended_Value("5.0")          = 5.0
X'Extended_Value("0.5E1")        = 5.0
X'Extended_Value("2#1.01#E2")    = 5.0
```

'Extended_Digits

Usage:

X'Extended_Digits(Base)

Returns the number of digits using base in the mantissa of model numbers of the subtype X.

Parameter Descriptions:

Base	The base that the subtype is defined in. If no base is specified, the default (10) is assumed. *Optional*

Examples:

Suppose the following type were declared:

```
type X is digits 5 range -10.0 .. 16.0;
```

Then the following would be true:

```
X'Extended_Digits     = 5
```

3.8.2.2.4. Fixed Point Attributes

'Extended_Image

Usage:

X'Extended_Image(Item,Fore,Aft,Exp,Base,Based)

Returns the image associated with Item as defined in Text_IO.Fixed_IO. The Text_IO definition states that it outputs the value of the parameter Item as a decimal literal with the format defined by the other parameters. If the value is negative, a minus sign is included in the integer part of the value of Item. If Exp is 0, the integer part of the output has as many digits as are needed to represent the integer part of the value of Item or is zero if the value of Item has no integer part. (LRM 14.3.8:13, 14.3.8:15)

For a prefix X that is a discrete type or subtype: this attribute is a function that may have more than one parameter. The parameter Item must be a Real value. The resulting string is without underlines or trailing spaces.

Parameter Descriptions:

Item	The item for which you want the image: it is passed to the function. *Required*
Fore	The minimum number of characters for the integer part of the decimal representation in the return string. This includes a minus sign if the value is negative and the base with the '#' if based notation is specified. If the integer part to be output has fewer characters than specified by Fore, leading spaces are output first to make up the difference. If no Fore is specified, the default value (2) is assumed. *Optional*
Aft	The minimum number of decimal digits after the decimal point to accommodate the precision desired. If the delta of the type or subtype is greater than 0.1, then Aft is 1. If no Aft is specified, the default (X'Digits-1) is assumed. If based notation is specified, the trailing '#' is included in Aft. *Optional*
Exp	The minimum number of digits in the exponent; the exponent consists of a sign and the exponent, possibly with leading zeros. If no Exp is specified, the default (3) is assumed. If Exp is 0, no exponent is used. *Optional*
Base	The base in which the image is to be displayed. If no base is specified, the default (10) is assumed. *Optional*
Based	An indication of whether you want the string returned to be in based notation or not. If no preference is specified, the default (false) is assumed. *Optional*

Examples:

Suppose the following type were declared:

```
type X is delta 0.1 range -10.0 .. 17.0;
```

Then the following would be true:

```
X'Extended_Image(5.0)                    = " 5.00E+00"
X'Extended_Image(5.0,1)                  = "5.00E+00"
X'Extended_Image(-5.0,1)                 = "-5.00E+00"
X'Extended_Image(5.0,2,0)                = " 5.0E+00"
X'Extended_Image(5.0,2,0,0)              = " 5.0"
X'Extended_Image(5.0,2,0,0,2)            = "101 0"
X'Extended_Image(5.0,2,0,0,2,True)       = "2#101.0#"
X'Extended_Image(5.0,2,2,3,2,True)       = "2#1.1#E+02"
```

'Extended_Value

Usage:

X'Extended_Value(Image)

Returns the value associated with Item as defined in Text_IO.Fixed_IO. The Text_IO definition states that it skips any leading zeros, reads a plus or minus sign if present, then reads the string according to the syntax of a real literal. The return value is that which corresponds to the sequence input. (LRM 14.3.8:9, 14.3.8:10)

For a prefix X that is a discrete type or subtype; this attribute is a function with a single parameter. The actual parameter Item must be of predefined type string. Any leading or trailing spaces in the string X are ignored. In the case where an illegal string is passed, a Constraint_Error is raised.

Parameter Descriptions:

Image	Parameter of the predefined type string. The type of the returned value is the base type of the input string. *Required*

Examples:

Suppose the following type were declared:

```
type X is delta 0.1 range -10.0 .. 17.0;
```

Then the following would be true:

```
X'Extended_Value("5.0")          = 5.0
X'Extended_Value("0.5E1")        = 5.0
X'Extended_Value("2#1.01#E2")    = 5.0
```

'Extended_Fore

Usage:

X'Extended_Fore(Base.Based)

Returns the minimum number of characters required for the integer part of the based representation of X.

Parameter Descriptions:

Base	The base in which the subtype is to be displayed. If no base is specified, the default (10) is assumed. *Optional*
Based	An indication of whether you want the string returned to be in based notation or not. If no preference is specified, the default (false) is assumed. *Optional*

Examples:

Suppose the following type were declared:

```
type X is delta 0.1 range -10.0 .. 17.1;
```

Then the following would be true:

```
X'Extended_Fore                    = 3  -- "-10"
X'Extended_Fore(2)                 = 6  -- " 10001"
```

'Extended_Aft

Usage:

X'Extended_Aft(Base,Based)

Returns the minimum number of characters required for the fractional part of the based representation of X.

Parameter Descriptions:

Base	The base in which the subtype is to be displayed. If no base is specified, the default (10) is assumed. *Optional*
Based	An indication of whether you want the string returned to be in based notation or not. If no preference is specified, the default (false) is assumed. *Optional*

Examples:

Suppose the following type were declared:

```
type X is delta 0.1 range -10.0 .. 17.1;
```

Then the following would be true:

```
X'Extended_Aft     = 1  -- "1" from 0.1
X'Extended_Aft(2)  = 4  -- "0001" from 2#0.0001#
```

3.8.3. Package System.

The current specification of package System is provided below.

```
with Unchecked_Conversion;

package System is

   --------------------------
   -- CUSTOMIZABLE VALUES
   --------------------------

   type Name     is (TeleGen2);

   System_Name   : constant name := TeleGen2;

   Memory_Size   : constant := (2 ** 31) -1; --Available memory, in storage units
   Tick          : constant := 2.0 / 100.0;   --Basic clock rate, in seconds

   type Task_Data is --Adaptation specific customization for task objects
       record
          null;
       end record;


   ------------------------------------------------------------
   -- NON-CUSTOMIZABLE, IMPLEMENTATION-DEPENDENT VALUES
   ------------------------------------------------------------

   -- See Table 3-2 for the values for attributes of
   -- types Float and Long_Float

   Storage_Unit : constant := 8;
   Min_Int      : constant := -(2 ** 31);
   Max_Int      : constant := (2 ** 31) - 1;
   Max_Digits   : constant := 15;
   Max_Mantissa : constant := 31;
   Fine_Delta   : constant := 1.0 / (2 ** Max_Mantissa);


   subtype Priority is Integer Range 0 .. 63;



   -------------------------
   -- ADDRESS TYPE SUPPORT
   -------------------------

   type Memory is private;
   type Address is access Memory;
     --
     -- Ensures compatibility between addresses and access types.
     -- Also provides implicit NULL initial value.
```

```
Null_Address: constant Address := null;
  --
  -- Initial value for any Address object


type Address_Value is range -(2**31)..(2**31)-1;
  --
  -- A numeric representation of logical addresses for use in address clauses


Hex_80000000 : constant Address_Value := - 16#80000000#;
Hex_90000000 : constant Address_Value := - 16#70000000#;
Hex_A0000000 : constant Address_Value := - 16#60000000#;
Hex_B0000000 : constant Address_Value := - 16#50000000#;
Hex_C0000000 : constant Address_Value := - 16#40000000#;
Hex_D0000000 : constant Address_Value := - 16#30000000#;
Hex_E0000000 : constant Address_Value := - 16#20000000#;
Hex_F0000000 : constant Address_Value := - 16#10000000#;
  --
  -- Define numeric offsets to aid in Address calculations
  -- Example:
  --   for Hardware use at Location (Hex_F0000000 + 16#2345678#);


function Location is new Unchecked_Conversion (Address_Value, Address);
  --
  -- May be used in address clauses:
  --
  --   Object: Some_Type;
  --   for Object use at Location (16#4000#);


function Label (Name: String) return Address;
pragma Interface (META, Label);
  --
  -- The LABEL meta-function allows a link name to be specified as address
  -- for an imported object in an address clause:
  --
  --  Object: Some_Type;
  --  for Object use at Label("OBJECT$$LINK_NAME");
  --
  -- System.Label returns Null_Address for non-literal parameters.




--=========================
-- ERROR REPORTING SUPPORT
--=========================

procedure Report_Error;
pragma Interface (Assembly, Report_Error);
pragma Interface_Information (Report_Error, "REPORT_ERROR");
```

```
   --
   -- Report_Error can only be called in an exception handler and provides
   --    an exception traceback like tracebacks provided for unhandled
   --    exceptions
   --

   --=================
   -- CALL SUPPORT
   --=================

    type Subprogram_Value IS
    record
       Proc_addr    : Address;
       Parent_frame : Address;
    end record;

     --
     -- Value returned by the implementation-defined 'Subprogram_Value
     -- attribute. The attribute is not defined for subprograms with
     -- parameters, or functions.


private
   type Memory is
   record
     null;
   end record;

end System;
```

3.8.3.1. System.Label

The System.Label meta-function is provided to allow users to address objects by a linker-recognized label name. This function takes a single string literal as a parameter and returns a value of System.Address. The function simply returns the run-time address of the appropriate resolved link name, the primary purpose being to address objects created and referenced from other languages.

- When used in an address clause, System.Label indicates that the Ada object or subprogram is to be referenced by a label name. The actual object must be created in some other unit (normally by another language), and this capability simply allows the user to import that object and reference it in Ada.

- When used in an expression, System.Label provides the link time address of any name: a name that might be for an object a subprogram, etc.

3.8.3.2. System.Report_Error

Report_Error can only be called from within an exception handler. This routine displays the normal exception traceback information to standard output. It is essentially the same traceback that could be obtained if the exception were unhandled and propagated out of the program, but the user may want to handle the exception and still display this information. The user may also want to use this capability in a user handler at the end of a task (since those exceptions will not be propagated to the main program). Note that the user can also get this capability for all tasks using the -X binder switch.

For details on the output, see the programming guide chapter in this volume, section "Exception Handling."